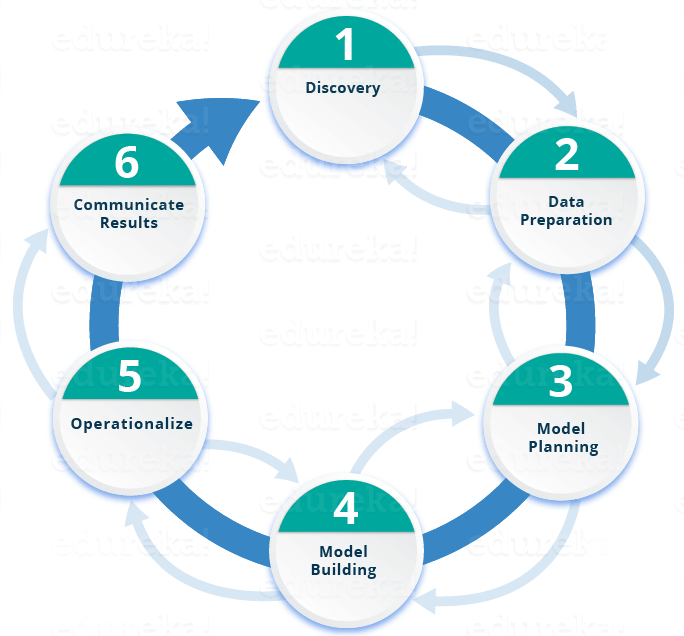
# Infrastructure pour le traitement des données

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)*.

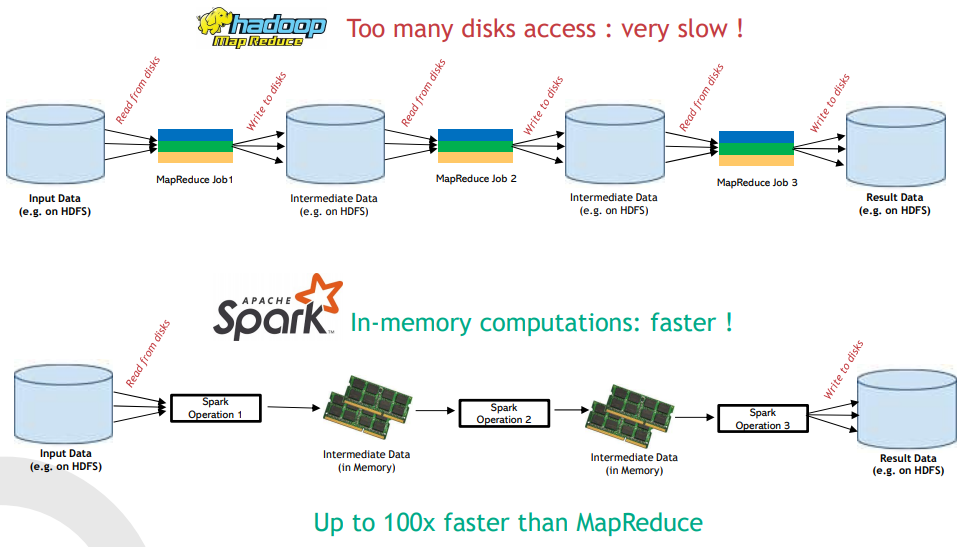
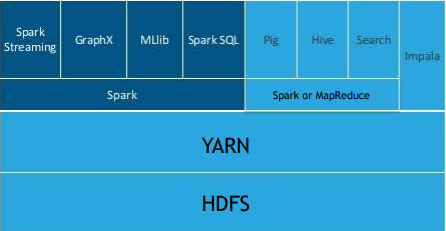
*Note au lecteur : L’ensemble du code python (ainsi que la configuration des différents nœuds Spark) qui nous a permis de mener à bien cette étude est disponible en fin de rapport.*

## 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)*

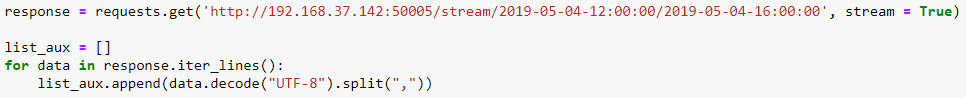
Notre cluster Spark contient un nœud master (driver node) et 7 nœuds travailleurs qui communique via un cluster manager. Le driver est responsable du déroulement du programme Spark : il crée le contexte et envoie des tâches aux nœuds travailleurs pour qu’ils les exécutent. Les nœuds travailleurs font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

# Analyse des données batch

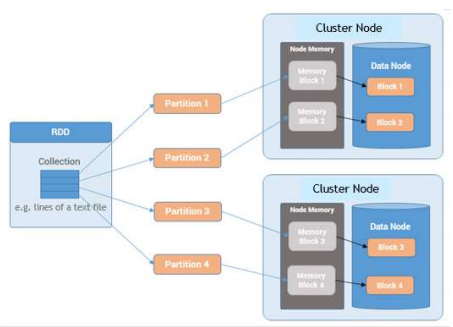
## Obtention des données grâce au 1er micro-service

## 

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

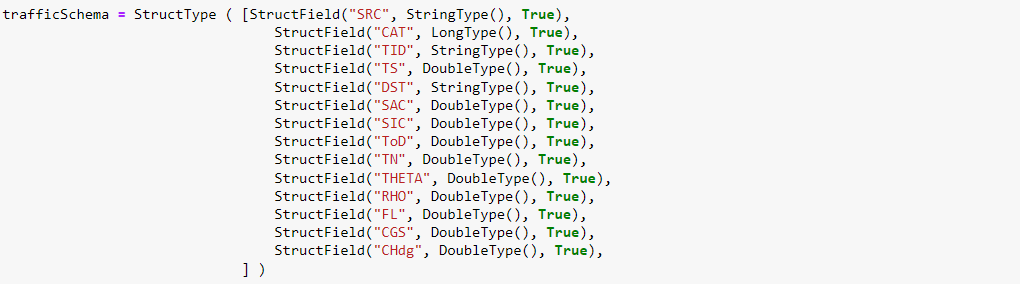
Dans la première partie de ce rapport, nous avions défini les champs présents dans les trames et ceux que nous souhaitions conservés pour analyse (resp. SRC\_MAC\_ADDRESS, CAT, TID, TS, DST, SAC, SIC, ToD, TN, THETA, RHO, FL, CGS, CHDG).

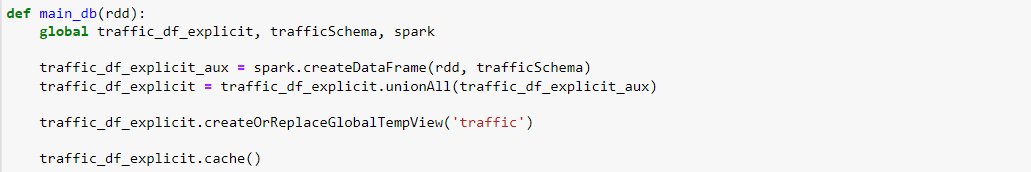
Avant analyse (requêtage, apprentissage, prédiction) et visualisation grâce à Spark SQL, nous avons souhaité apporter quelques modifications aux données reçues, notamment convertir au format numérique les variables quantitatives et remplacer toutes les valeurs manquantes par le même champ NULL. Tout 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.

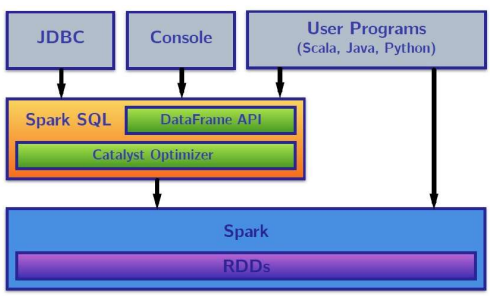
## Utilisation de Spark SQL et création d’une vue temporaire pour le requêtage

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.

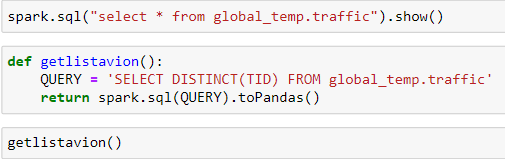


Ainsi, Spark SQL nous évite de nous poser des questions quant à l’ordre dans lequel nous devons utiliser les fonctions de mapping et de reducing (reduceByKey, groupByKey) puisqu’il optimise le code automatiquement en fonction de la requête SQL : il minimise tout seul le data shuffling.

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.

## Interface utilisateur pour le déroulement : 2e microservice

## Visualisation des données

Grâce à Spark SQL, nous avons ainsi pu effectuer des requêtes dont l’objectif est de renvoyer un sous-ensemble de notre base afin d’afficher les variables de chaque trame sur différents graphes.

Nous avons développé des fonctions de visualisation qui permettent à l’utilisateur, sur une période de temps donnée, de choisir entre : visualiser les données reçues par différents radars et un seul avion (i), visualiser différents paquets reçues de la part de plusieurs avions mais pour un seul radar (ii), visualiser un avion et un radar (iii). Ce dernier cas nous a été extrêmement utile pour réaliser des prédictions, détaillée dans la partie suivante pour le batch et dans le chapitre suivant pour le temps réel.

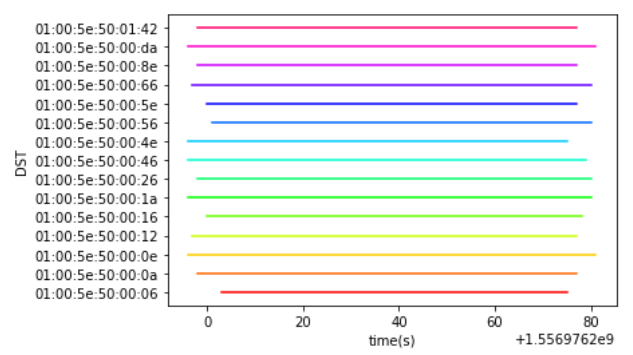
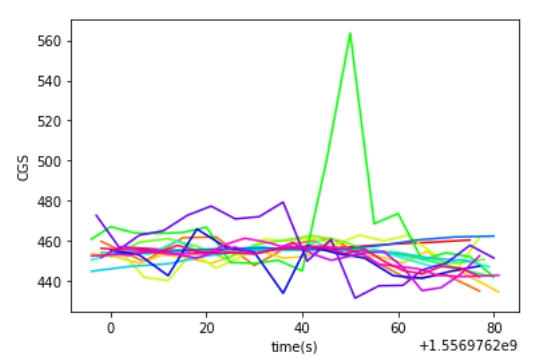
Tout cela est accessible via une architecture orientée services (voir figure ci-dessous).

Une image contenant table

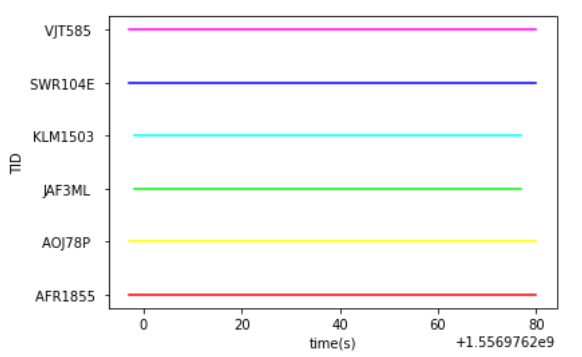
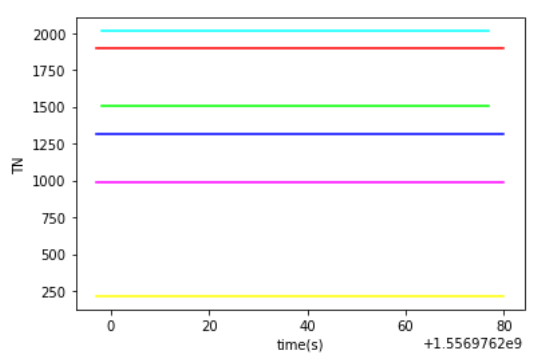
Description générée automatiquement

NB : Pour plus d’informations concernant les micro-services (dont celui-ci), vous pouvez consulter notre rapport d’Architecture Orientée Services qui sera disponible le 27/01/2021 à cette adresse :

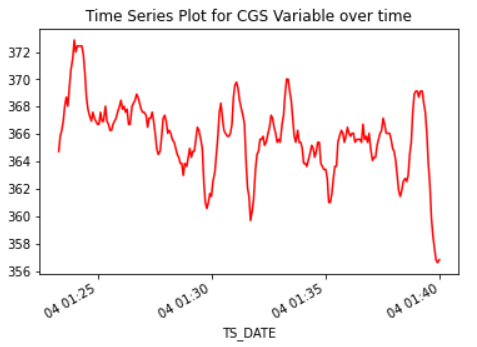
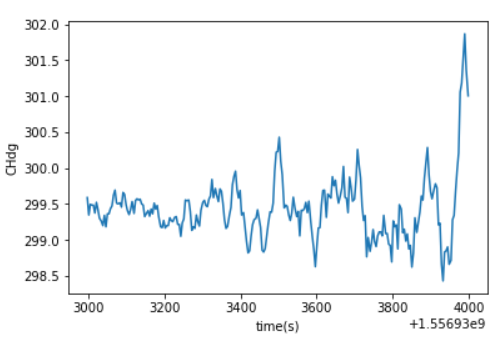
### Un Avion, x radars (au gré de l’utilisateur)



### Un Radar, x avions (au gré de l’utilisateur)



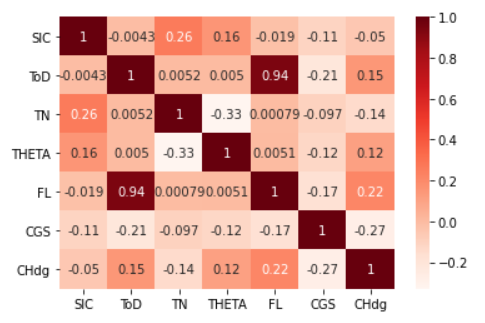
### Un Avion, un radar



## Prédiction : les séries temporelles à l’honneur

### Justification de l’utilisation du modèle de Machine Learning des séries temporelles

Après consultation du cloud fourni par ACTIVUS, nous avons souhaité élaborer des prédictions qui se concentraient plus sur la position de l’avion que sur le nombre de données reçues par les radars. En effet, des modèles de prédiction qui se concentraient sur les radars avaient déjà été réalisés (auto-encodeur notamment). Ainsi, nous avons pu prédire l’évolution de certains paramètres, en batch puis en temps-réel. Nous avons également détecté si les données reçues par l’avion étaient pertinentes ou si elles étaient faussées (mauvais modèle ou, à défaut, mauvaise information de la part des capteurs).



Mais avant cela, en plus des différents graphes (correspondant aux différentes variables étudiées), il nous a été possible d’étudier la corrélation entre chacune des variables. Il était nécessaire de le faire dans la mesure où ces corrélations allaient nous aiguiller vers le modèle de machine learning à utiliser. En effet, dans le cas d’une forte corrélation entre les variables, nous aurions pu les étudier davantage, entrainer un modèle de régression et prédire grâce à l’équation obtenue.

Ici, on voit clairement que la corrélation entre les variables (ne pas tenir compte de ToD qui correspond au temps) n’est pas forte et donc que les modèles de régression ne seraient pas utiles.

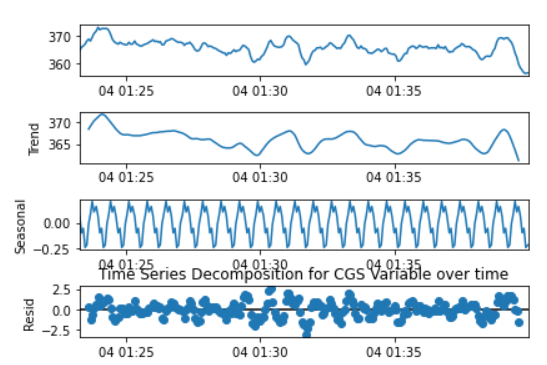
Mais, nous n’en restons pas là. En effet, nos données font partie d’une classe de problème de Machine Learning bien particulier : elles associent dépendance entre les données et temporalité (Panel Data).

C’est en ce sens que nous avons choisi d’utiliser le modèle de Machine Learning des séries temporelles.

### Description du modèle de séries temporelles

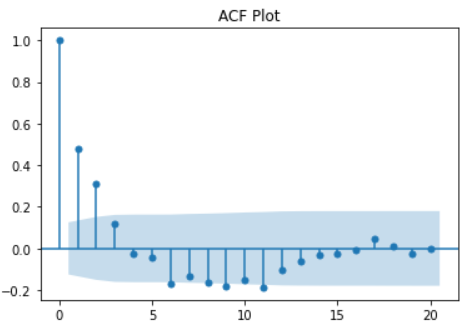
Dans notre cas, trois variables ont été mis à l’honneur dans notre étude des séries temporelles : CGS, CHdg et FL. Nous avons observé ces variables pour t=1, t=2, t=3, …, t=T et nous avons construit un modèle statistique pour prédire les valeurs suivantes à t=T+1, t=T+2, t=T+3.

Comme nous l’avons dit, ce qui fait la particularité des séries temporelles est que nous n’allons pas prédire la valeur de notre variable Y (ici *CGS*, *CHdg* ou *FL*) grâce à la dépendance de d’autres variables X comme cela peut être le cas pour des modèles de régression classique. Ainsi nous n’avons pas besoin de connaître les valeurs de variables X pour prédire Y. Seul la fonction Y et ses valeurs passées nous permettent de prédire des valeurs futures. Cette analyse peut paraître grossière car une étude de fonction ne suffit pas pour prédire une valeur future. Néanmoins, nous avons ici une première estimation qui, assez rapidement, peut donner un ordre d’idée des valeurs futures approximatives que l’on doit obtenir.

Passons à la description d’une série temporelle. On peut la décomposer en une tendance, une saisonnalité et une composante aléatoire. Il existe des séries dites additives, qui se décompose comme la somme des 3 composantes présentes, et des séries multiplicatives, qui correspond au produit de ces trois dernières. Dans notre cas, puisque nos données n’ont pas une variance qui augmente dans le temps, le choix de modèles additifs paraît pertinent pour décomposer la série.

Essentiellement, la tendance et la saisonnalité sont facilement déterminables mathématiquement. C’est plus complexe pour le bruit et nous devons appliquer une méthodologie stricte, divisée en plusieurs étapes pour obtenir une première approximation : il s’agit de la méthodo Box & Jenkins qui, de l’obtention d’une série stationnaire nous amène à la découverte des modèles ARMA.

## Séries stationnaires et modèles ARMA

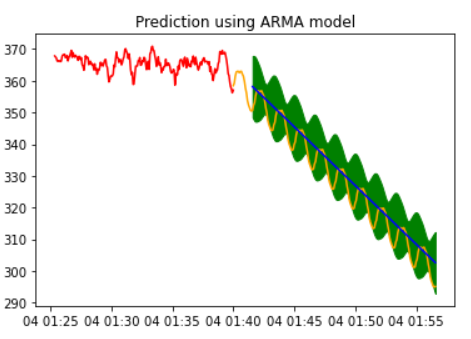
Les processus faiblement stationnaires doivent restent constants en moyenne et en variance et la covariance entre Y(t) et Y(t-s) ne doit dépendre que de la différence t en t-s donc de la période (cf. cours de Signaux Aléatoires, 3 MIC-IR). En utilisant le test statistique KPSS, nous avons obtenu que notre composant aléatoire n’était pas stationnaire. Pour pouvoir utiliser la modélisation ARMA, nous devons avoir un processus stationnaire. Le moyen le plus efficace pour ‘stationariser’ un processus est de différence Y(t) avec Y(t-1) pour enlever la tendance puis de prendre la différence entre Y(t) et Y(t-s) pour enlever la saisonalité. Une fois le processus stationnaire, il est possible de déterminer l’équation du processus en regardant la fonction d’autocorrélation (ACF) et la fonction d’autocorrélation partielle (PACF). Un processus stationnaire ARMA a une composante ACF et PACF qui décroit exponentiellement dans le temps. De l’étude de ces fonctions, il nous est possible de choisir parmi les différents modèles ARMA (AR, MA, ARMA, SARMA, ARIMA, SARIMA et SARIMA multiplicatives), une équation tenant plus ou moins compte des composants précédents et accentuant, ou non, le nombre de coefficients.

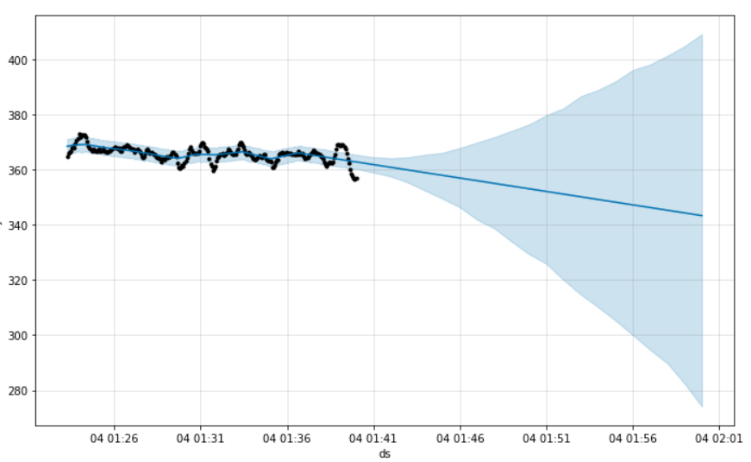
Pour s’éviter ce travail fastidieux, l’utilisation de la fonction ci-dessous nous permet d’avoir directement le meilleur modèle possible ARMA en choisissant un critère de référence (AIC, SBIC, etc)

### Premières estimations et premières prédictions

Nous cherchions un modèle mathématique pour caractériser notre série temporelle afin de prédire ses valeurs futures sans avoir de connaissance sur sa dépendance avec d’autres variables temporelles existantes. Pour cela, nous avons décomposer notre série et stationariser le bruit afin de déterminer le nombre de coefficients AR et MA.

Nous sommes maintenant en mesure de valider le modèle et prédire des valeurs futures ainsi qu’un intervalle de confiance associé. Nous avons donc sur le même graphe à la fois les valeurs précédentes qui nous ont servi à déterminer le modèle ainsi que les valeurs futures prédites.





Finalement, ce modèle, fastidieux mais simple, nous donne une première estimation des valeurs futures que doit prendre notre série avant d’utiliser des modèles plus complexes comme le deep learning et les réseaux de neurones. Par comparaison, nous avons également utilisé le package Fbprophet capable lui aussi de fournir une approximation des valeurs futures à obtenir pour cette même série. Il nous permet finalement d’obtenir un comportement sensiblement identique à notre modèle ARMA.

Puisqu’il permet une automatisation plus rapide de notre prédiction et nous fourni une liste de valeurs prédites plus rapidement, nous avons choisi de continuer avec ce modèle pour la prédiction en temps réel. Autre point très important, même s’il existe plusieurs modèles de machine learning sur MLLIB, bibliothèque pour faire tourner des modèles directement sur Spark, le modèle ARMA n’existe pas dans cette bibliothèque, ce qui complexifie la tâche si nous souhaitions l’utiliser directement sur SPARK.

## Bilan de la partie Batch

Nous avons maintenant une architecture fonctionnelle pour une utilisation en mode batch de nos données. Nous aurions pu étudier plus en détail les données en batch : faire des requêtes sur une plus longue période, faire des analyses plus complètes sur la totalité des données. En effet, grâce à l’utilisation de Spark, notre infrastructure est scalable et peut analyser des volumes de données bien plus conséquents.

Plutôt que d’entamer cette étude, nous avons choisi de nous challenger et faire fonctionner cette même architecture en temps réel. Le chapitre suivant explique, en détails, comme nous avons adapté infrastructure Spark, découverte et clean des données, apprentissage et prédiction pour un fonctionnement en temps réel.

C’est maintenant que l’étude de ces données prend tout son sens. Nous pouvons ainsi détecter des anomalies alors même que l’avion est encore en vol.

# Analyse des données en temps réel avec Spark

## Adaptation du 1er micro-service à une utilisation temps-réel

Cette fois-ci, nous allons demander au service de nous fournir les données en temp-réel. Pour ce cas d’exemple, nous simulerons le temps réel à partir d’une période déterminée. Pour l’utilisateur, sur du véritable temps-réel ou du temps-réel simulé, il pourrait être intéressant d’obtenir, à partir du 2e micro-service, une analyse en mode différé sur une période de temps pour obtenir les avions en vol, sélectionner un ou plusieurs avions, un ou plusieurs radars puis de lancer le temps-réel qui lui démarrerait à la suite de la période analysée en mode différé.

## Adaptation des RDD pour l’analyse de données temps réel

Par rapport à notre utilisation en mode différé, nous devons paralléliser les données au fur et à mesure.

Nous avons donc choisi, arbitrairement, de stocker et paralléliser les données sur Spark au bout d’un certain temps.

## Découverte et Nettoyage des données en temps réel

Par rapport à notre utilisation en mode différé, nous devons traiter les données au fur et à mesure.

Cela signifie que le clean des données et l’insertion sur Spark depuis une requête au 1er micro-service ne peut se faire après réception de l’intégralité des données. En effet, il n’y a pas de fin. Nous avons donc choisi, arbitrairement, de nettoyer les données au bout d’un certain temps, la fonction de nettoyage étant identique à celle utilisée en mode batch puisque les données proviennent de la même source et peuvent donc être traitées de la même manière.

## Utilisation de Spark SQL en temps réel

De même, la vue SQL créée sur les partitions Spark reste la même à la différence que nous venons arbitrairement rajouter un certain nombre de données sur les nœuds Spark au-fur-et-à-mesure. Nous réactualisons la vue à chaque insertion. Cela nous permet de requêter sur une ‘table’ qui est toujours actualisée au courant du temps.

## Prédiction des données en temps-réel

### Complexité de la prédiction temps-réel

Comme nous l’avions évoqué dans la partie précédente, la prédiction se fera à l’aide du package Fbprophet car la computation nous a semblé la plus rapide (par rapport au modèle ARMA) et qu’il nous permet d’obtenir rapidement un intervalle de confiance [yhat\_lower ; yhat\_upper] dans lequel nous allons recevoir nos données ainsi qu’une tendance approximative yhat.

Pour la prédiction en temps-réel, les choses se compliquent puisque nous devons prédire des valeurs futures mais cette prédiction de quelques valeurs doit arriver en base et être visualisée avant que les données temps réel ne s’insère elles-mêmes en base. Cela veut dire que le modèle doit être suffisamment efficient et rapide pour ne pas se faire dépasser par les données temps réel qui suivent la même cadence malgré la computation du modèle.

### Complexité de la prédiction temps-réel

Le point précédent nous amène à effectuer deux choix : la computation de notre modèle directement sur les nœuds Spark avant requêtage et le threading du requêtage et de l’envoi des données prédites en base afin de ne pas surcharger le buffer qui réceptionne les données temps réel.



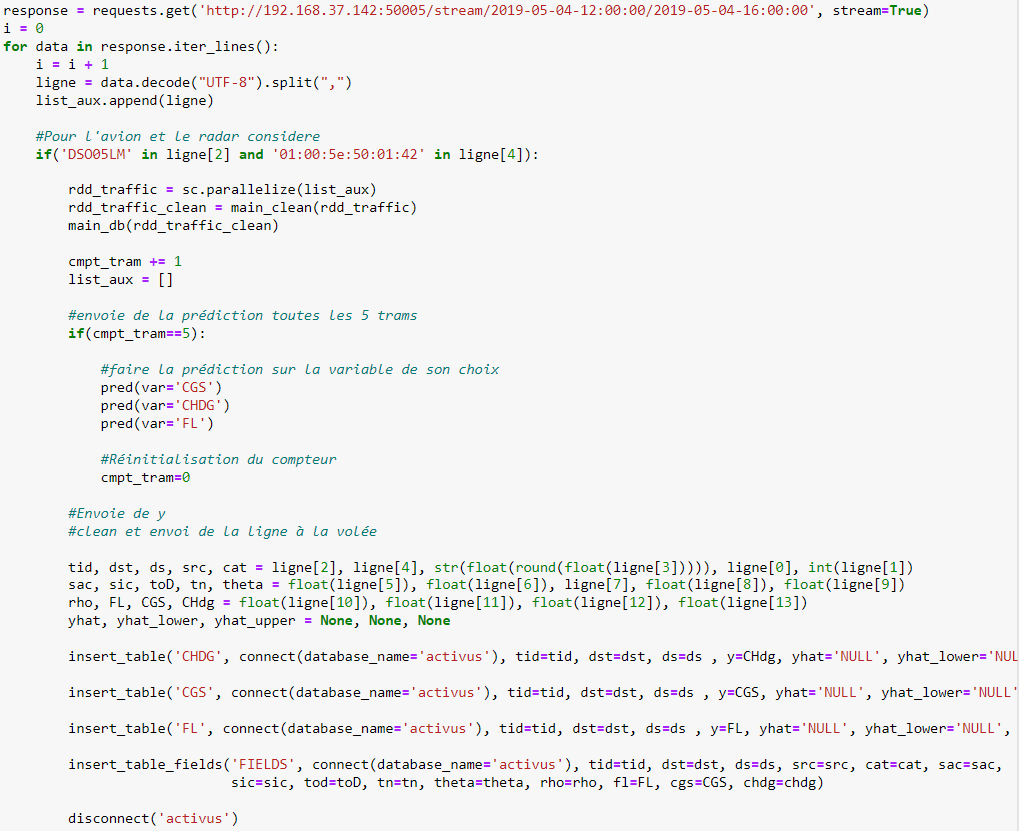
## Insertion en temps réel dans une base de données SQL

Parce que l’intégration de Grafana et Spark semblait plus difficile à obtenir dans le peu de temps que nous avions, nous avons choisi d’insérer dans une base de données SQL, sur une VM, les données traitées.

Il y avait donc plusieurs types d’insertion, depuis notre application python, en temps réel :

* Insertion des données traitées à la volée pour une visualisation en temps réel. Ces données ne contiennent aucune prédiction ;
* Insertion des premières données prédites. Ces données ne contiennent aucune valeur pour le y réel ;
* Update des anciennes prédictions avec les prédictions actualisées. En effet, le modèle fonctionne sur toutes les données reçues en temps réel, plus le modèle contient de données, plus il s’affine jusqu’à arriver à de l’overfitting.
* Insertion de nouvelles prédictions pour les données les plus récentes

## Scénario démonstratif utilisé



## Visualisation des données réelles et prédites en temps-réel sur Grafana