CentraleSupélec

eDF

CentraleSupélec en partenariat avec EDF

Solution Big Data sur les solutions de mobilité en Ile-de-France

Rapport de projet – Etudes du trafic sur la ligne H

Roméo DJOUKENG, David TIA, Etienne VINCENT
20/03/2020

1) Introduction

Ce projet porte sur l'analyse des données en temps réel des bus, tramways, métros, RER ou trains d'Ile-de-France. Pour ce projet, notre groupe a choisi de se concentrer sur les données de **la ligne H.** Les données sont mises à disposition par Mobilité Ile-de-France au moyen d'une API accessible depuis leur portail : https://portal.api.iledefrance-mobilites.fr/.

Le Github du projet est accessible au lien suivant : https://github.com/DavidQuarz/Big_Data.git

2) Architecture envisagée

Pour réaliser récupérer les données en temps réel et les exploiter, nous avons prévu l'architecture suivante :

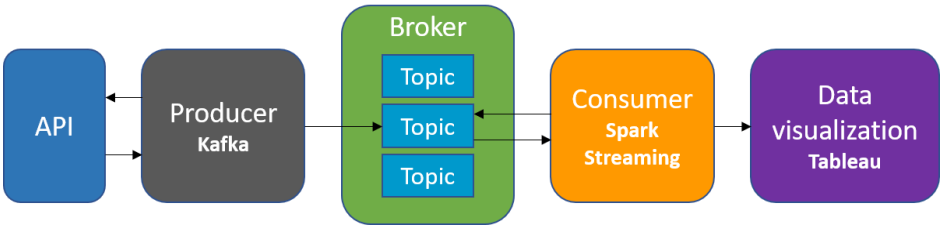


Figure 1: Architecture

1ère étape - Connexion à l'API : Avant de pouvoir récupérer les données, il nous est nécessaire de nous authentifier (*credentials*) sur le site de mobilité Ile-de-France avant chaque requête à l'API. Ceci fait, un *token* nous est attribué pour requêter l'API.	**4ème étape - Enrichissement des données :** Les données sont lues depuis le *Topic* par le *Consumer* puis enrichies par différentes opérations pour être ensuite exploitées. Le *Consumer* est un script python utilisant Spark Structured Streaming.
2ème étape - Récupération des données temps réel : L'API nous permet de récupérer les données en temps réel. Notre *Producer*, un script python récupère périodiquement les données et les écrits sur un topic Kafka.	**5ème étape - Exploitation des données :** Les données sont exploitées avec *Tableau Software*. Ce dernier se connecte à Spark Thrift Server afin de pouvoir requêter les tables issues du *consumer*.
3ème étape - Stockage de données brutes : Les données extraites par le *Producer* sont écrites sur un *Topic* du *Broker*.	

3) Connexion à l'API

Pour se connecter à l'API, nous utilisons une fonction python *get_access_token()* qui permet d'une part de s'authentifier lors de la connexion à l'API (*client_id* et *client_secret*) et d'autre part de demander un *token* pour pouvoir requêter l'API.

```python
def get_access_token():
  urlOAuth= 'https://as.api.iledefrance-mobilites.fr/api/oauth/token'
  #client_id='342fe700-3f5c-417b-a36d-14b063461ecd'
  #client_secret='58032e87-8a3c-4d52-ac68-607dbca615e2'
  client_id='1e090371-7332-4cd0-89e7-fe5b910e8206'
  client_secret='7bd05965-8e19-4d3f-b105-ad7783381a6a'
  data =dict(
    grant_type='client_credentials' ,
    scope='read-data',
    client_id=client_id,
    client_secret=client_secret)
  response= requests.post(urlOAuth, data=data, verify=False)

  if(response.status_code != 200):
    print('Status: ', response.status_code ,'Erreur sur la requete; fin de programme')
    exit()
  jsonData= response.json()
  return jsonData['access_token']
```

Figure 2: Producer - Récupération du token

4) Récupération des données temps réel

La récupération des données se fait avec un *Producer* écrit en python. Celui-ci récupère les données de la ligne H toutes les 30 secondes et les stocke sur le *topic « RealTime-Transilien-Topic »*. Ci-dessous la chronologie des tâches du *Producer :*

a) Création d'un *Producer* Kafka sur la machine local sur le port 9092,
b) Demande d'un *token* à l'API,
c) Récupération des données temps réel,
d) Conversion au format JSON,
e) Stockage des données sur le *topic « RealTime-Transilien-Topic »*
f) Après 30 secondes, répétition de la tâche c) et d).

```python
import requests, json, time
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers="localhost:9092")
defaultToken=get_access_token()
arrets=["STIF:StopPoint:Q:41123:"]
temp=["STIF:StopPoint:Q:41107:","STIF:StopPoint:Q:41111:","STIF:StopPoint:Q:41113:",
session=requests.session()
returndata='{"data":['
i=0
while True:
  for arret in arrets:
    response=getStationData(session,arret,defaultToken)
    i=i+1
    print(response.status_code)
    if(response.status_code==401):
      defaultToken=get_access_token()
      response=getStationData(session,arret,defaultToken)
    data=response.json()
    returndata=returndata+json.dumps(data)
    if(i<len(arrets)):
      returndata=returndata+','
  returndata=returndata+']}'
  producer.send("RealTime-Transilien-Topic", returndata)
  print('batch sent to Topic')
  returndata='{"data":['
  i=0
  time.sleep(30)
```

Figure 3: Producer - Réquête à l'API

```
def getStationData(session,idStation,defaultToken):
  URL="https://traffic.api.iledefrance-mobilites.fr/v1/tr-unitaire/stop-monitoring"
  PARAMS={}
  headerParams={}
  PARAMS["MonitoringRef"]=idStation
  headerParams["Authorization"]= "Bearer "+ defaultToken
  response=session.get(URL,params=PARAMS, headers=headerParams)
  return response
```

Figure 4 : Prodcuer - Récupération des données

Le *token* étant valide pour une durée limitée, un nouveau est redemandé dès qu'il n'est plus valide.

5) Stockage de données brutes

L'ensemble des données temps réel sont stocké sur le *topic « RealTime-Transilien-Topic »* au niveau de la machine local. Les données sont stockées sous forme de *stream*.

6) Traitement et enrichissement des données

L'enrichissement des données se fait sur la base des données du stream lu sur le *topic*. Nous utilisons Spark Structured Streaming pour traiter et enrichir ce *stream* de données. Spark structured Streaming n'étant pas capable de lire la structure de données d'un *json*, nous avons au préalable enregistré sur HDFS un exemplaire d'un fichier json pour en déduire son schéma. Nous demandons à Spark d'inférer le schéma de cet exemplaire et l'enregistrons dans une variable *schema*. Ceci est possible car le retour de l'API est au format JSON et que Spark sait inférer le schéma d'un JSON.

```
#Construction du contexte spark
spark = SparkSession.builder \
  .appName("Embedding Spark Thrift Server") \
  .config("spark.sql.hive.thriftServer.singleSession", "True") \
  .config("hive.server2.thrift.port", "10001") \
  .config("javax.jdo.option.ConnectionURL", \
          "jdbc:derby:;databaseName=metastore_db2;create=true") \
  .enableHiveSupport() \
  .getOrCreate()

#Chargement du schema template
schema=spark.read.json("hdfs:///tmp/ttempJson.json",multiLine=True).schema

#Chargement du topic depuis Kafka
s1= spark.readStream.format("kafka")\
  .option("kafka.bootstrap.servers", "localhost:9092")\
  .option("subscribe","RealTime-Transilien-Topic").load()
```

Figure 5: Consumer - Création de la session Spark

En partant du postulat que tous les *stream* enregistrés sur le *topic* auront la même structure de données, nous précisons à Spark Streaming de lire pour chaque nouveau *stream* le contenu de la colonne value et d'y associé le schéma enregistré dans la variable *schema*. Nous parvenons de cette manière à déterminer la structure de données de chaque stream avec Spark structured Streaming. A partir de ce *stream*, nous effectuons diverses opérations afin de traiter et d'enrichir les données.

```
#Parsing et Mise en forme du Json Stream
jsondf=s1.selectExpr("CAST(value as STRING)","CAST(timestamp as TIMESTAMP)")

jsondf=jsondf.select(from_json(col("value"),schema),"timestamp")
jsondf=jsondf.select("jsontostructs(value).data","timestamp")
jsondf=jsondf.select(explode("data"),"timestamp")
jsondf=jsondf.select("col.Siri.ServiceDelivery.StopMonitoringDelivery.MonitoredStopVisit","timestamp")
jsondf=jsondf.select(explode("MonitoredStopVisit"),"timestamp")
jsondf=jsondf.select(explode("col"),"timestamp")

jsondf=jsondf.select("col.MonitoredVehicleJourney.FramedVehicleJourneyRef.DatedVehicleJourneyRef",\
                     "col.MonitoringRef.value",\
                     "col.MonitoredVehicleJourney.MonitoredCall.StopPointName",\
                     "col.MonitoredVehicleJourney.MonitoredCall.DestinationDisplay",\
                     "col.MonitoredVehicleJourney.MonitoredCall.AimedDepartureTime",\
                     "col.MonitoredVehicleJourney.MonitoredCall.ExpectedDepartureTime","timestamp")

jsondf=jsondf.withColumn("StopPointName", explode("StopPointName"))
jsondf=jsondf.withColumn("DestinationDisplay", explode("DestinationDisplay"))
jsondf=jsondf.select("DatedVehicleJourneyRef",\
                     col("value").alias("StopPoint"),\
                     col("StopPointName.value").alias("StopPointName"),\
                     col("DestinationDisplay.value").alias("DestinationDisplay"),\
                     "AimedDepartureTime","ExpectedDepartureTime","timestamp")

#Convertion des dates au format unix timestamp pour pouvoir faire la difference
jsondf=jsondf.withColumn('converted_aimed',unix_timestamp(col('AimedDepartureTime'), "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"))
jsondf=jsondf.withColumn('converted_expected',unix_timestamp(col('ExpectedDepartureTime'), "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"))
jsondf=jsondf.withColumn('attente',col('converted_expected')-col('converted_aimed'))

#Mise en place d'un watermark
windowedjsondf= jsondf.withWatermark("timestamp", "10 seconds")\
                      .groupBy(window(col("timestamp"), "60 minutes","60 minutes"),\
                               jsondf.DatedVehicleJourneyRef ,\
                               jsondf.StopPoint,\
                               col("StopPointName"),\
                               col("DestinationDisplay"))\
                      .agg(max('attente').alias('attente'))

windowedjsondf=windowedjsondf.selectExpr("CAST(window.start as STRING)",\
                                         "CAST(window.end as STRING)",\
                                         "DatedVehicleJourneyRef",\
                                         "StopPoint","StopPointName","DestinationDisplay","attente")
```

Figure 6: Consumer - Traitement du stream de données

7) Stockage des données enrichies

a) Solution retenue : Stockage dans Spark Thrift Server (memory + csv)

Une fois enrichies, les données sont stockées afin qu'elles soient exposées au logiciel *Tableau Software*.

Idéalement, notre *consumer* stocke les données issues du *stream* au format "*memory*" sur *Spark Thrift Server* (*STS*). Ces données sont ensuite récupérées par Tableau Software au moyen de *STS* qui expose les données. En raison de problèmes de compatibilité des logiciels à notre disposition, Tableau Software ne parvient pas à lire les données au format "*memory*" (table temporaire) depuis *STS*.

Pour contourner ce problème, nous stockons toujours la sortie du *consumer* dans *STS* au format "*memory*" dans une table **temporaire**. Nous créons en parallèle dans la même instance *STS* une autre table **permanente** au format "csv" avec l'outil Beeline. Les données de table temporaire seront transférées vers la table permanente pour ensuite être exposée à Tableau Software.

```
68  #Création d'un table temporaire
69  windowedjsondf.createGlobalTempView("ratp")
70
71  """ LAUNCH STS """
72  java_import(spark.sparkContext._gateway.jvm, "")
73  spark.sparkContext._gateway.jvm.org.apache.spark.sql.hive.thriftserver \
74    .HiveThriftServer2.startWithContext(spark._jwrapped)
75  """ STS RUNNING """
76
77  query=windowedjsondf.writeStream\
78      .outputMode("complete")\
79      .format("memory")\
80      .trigger(processingTime='2 minutes')\
81      .queryName("ratp")\
82      .option("truncate",False)\
83      .start()
84  query.awaitTermination()
```

Figure 7 : Consumer v2 - Lancement et écriture des données dans STS

```
#Lancement de Beeline
/usr/hdp/current/spark2-thriftserver/bin/beeline -u jdbc:hive2://localhost:10001
```

```
#Création de la table permanente "transilien" dans Beeline (à faire une seule fois)
CREATE EXTERNAL TABLE IF NOT EXISTS transilien(
  `start` STRING,
  `end` STRING,
  `DatedVehicleJourneyRef` STRING,
  `StopPoint` STRING,
  `StopPointName` STRING,
  `DestinationDisplay` STRING,
  `attente` BIGINT
    )
  COMMENT 'Transilien data'
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  STORED AS TEXTFILE
  LOCATION '/user/root/transilien';
```

```
#Lancement du consumer consumerJob_thrift_v2.py
spark-submit \
```

```
--packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0 \
--conf spark.sql.hive.thriftServer.singleSession=true \
  consumerJob_thrift_v2.py
```

Pour que la table permanente soit mise à jour périodiquement, avec la commande *crontab*, nous avions planifié un job qui se lance toutes les 2 minutes et écrase les données de la table permanente avec les données de la table temporaire. Ainsi toutes les deux minutes on pourra avoir un rafraichissement des données dans tableau.

```
#Création d'une tâche planifiée
crontab -e

> */2 * * * * /usr/hdp/current/spark2-thriftserver/bin/beeline \
>        -u jdbc:hive2://localhost:10001 \
>        --outputformat=csv2 \
>        -e "INSERT OVERWRITE TABLE transilien SELECT * FROM ratp;"
```

Tableau Software étant capable de lire les tables permanentes depuis *STS*, nous parvenons ainsi à récupérer les données issues du *consumer* et à les exploiter.

Avant d'être parvenu à cette solution, nous avions envisagé deux autres solutions que nous avons abandonnées.

b) Solutions envisagées et abandonnées

Stockage dans Thrift (memory)

Cette solution a été envisagée et permet de stocker dans l'instance *STS,* l'*output* du *consumer* au format "*memory*". En raison de problème de compatibilité entre logiciels, Tableau Software ne parvient pas lire les tables temporaires et donc le format "*memory*". Cette solution a été abandonnée. Il est a noté que ce problème a été résolu dans les versions postérieures des logiciels que nous utilisons actuellement.

```
54  """ LAUNCH STS """
55  java_import(spark.sparkContext._gateway.jvm, "")
56  spark.sparkContext._gateway.jvm.org.apache.spark.sql.hive.thriftserver \
57    .HiveThriftServer2.startWithContext(spark._jwrapped)
58  """ STS RUNNING """
59
60  query=windowedjsondf.writeStream\
61      .outputMode("update")\
62        .format("memory")\
63      .trigger(processingTime='2 minutes')\
64        .queryName("ratp")\
65        .option("truncate",False)\
66      .start()
67  query.awaitTermination()
```

Figure 8 : Consumer v0 – Lancement et écriture des données dans STS

```
#Lancement du consumer consumerJob_thrift_v0.py
spark-submit \
--packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0 \
--conf spark.sql.hive.thriftServer.singleSession=true \
  consumerJob_thrift_v2.py
```

Stockage dans Thrift (memory) + Persistence dans HIVE

Cette solution a été envisagée et permet de stocker dans l'instance *STS,* l'*output* du *consumer* au format "*Hive*". Nous créons manuellement en parallèle une table sur HIVE et nous faisons persister les données au format "*Hive*" dans la table créée dans HIVE.

```
#Création de la table "transilien" sur Data Analytics Studio (à faire une seule fois)
CREATE TABLE `default`.`transilien` (
  `start` STRING,
  `end` STRING,
  `DatedVehicleJourneyRef` STRING,
  `StopPoint` STRING,
  `StopPointName` STRING,
  `DestinationDisplay` STRING,
  `attente` BIGINT
)
```

Cette solution permet à Tableau Software d'accéder aux données depuis HIVE. Néanmoins, en raison du temps de requêtage trop long du fait de l'écriture en permanence dans la table nous avions abandonné cette solution qui nous permettait quand même d'avoir les données dans Tableau.

```
query=windowedjsondf.writeStream\
        .format(HiveWarehouseSession.STREAM_TO_STREAM)\
      .outputMode("update")\
      .trigger(processingTime='2 minutes')\
        .option("metastore", "thrift://localhost:9083")\
        .option("db", "default")\
        .option("table", "transilien")\
      .option("checkpointLocation", "checkpoint_stream")\
      .start()
query.awaitTermination()
```

Figure 9 : Consumer v1 - Ecriture dans la table HIVE

```
#Lancement du consumer consumerJob_thrift_v1.py
spark-submit\
  --jars /usr/hdp/3.0.1.0-187/hive_warehouse_connector/hive-warehouse-connector
assembly-1.0.0.3.0.1.0-187.jar \
  --py-files /usr/hdp/3.0.1.0-187/hive_warehouse_connector/pyspark_hwc
1.0.0.3.0.1.0-187.zip \
  --conf spark.security.credentials.hiveserver2.enabled=false\
  --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0   consumerJob_
thrift_v1.py
```

8) Bonus : Estimer la position d'un train

Les données qui serviront à traiter cette requête sont enrichies et stockées de la même manière qu'en partie 6) et 7) :

L'approche utilisée est la suivante :

Avec l'API nous sommes à même de récupérer à un instant donné pour chaque gare les temps de prochain passage. Si nous stockons ces données sur la durée il est donc possible en se basant sur les horaires de passage passées de déterminé la dernière gare traversée par le train et la prochaine gare.

Ces requêtes sont explicites dans la jointure faite dans tableau pour avoir les données finaux

```
#traitement additionnel pour la prediction de localisation
#on garde pour chaque station l'heure de depart expected le plus a jour
passagedf=jsondf.groupBy(jsondf.DatedVehicleJourneyRef ,\
                         jsondf.StopPointName,jsondf.DestinationDisplay)\
                               .agg(max('converted_expected')\
                               .alias('converted_expected'),\
                         max('converted_expected_arrival').alias('converted_expected_arrival'))
passagedf.createGlobalTempView("passage")
```

Figure 10: Traitement du stream de données (Bonus)

```
query1=passagedf.writeStream\
    .outputMode("complete")\
      .format("memory")\
    .trigger(processingTime='1 minutes')\
      .queryName("passage")\
      .option("truncate",False)\
    .start()
query.awaitTermination()
```

Figure 11: Consumer v2 - Ecriture des données dans STS (bonus)

```
#Création de la table permanente "passages" dans Beeline (à faire une seule fois)
CREATE EXTERNAL TABLE IF NOT EXISTS passages(
   `DatedVehicleJourneyRef` STRING,
   `StopPointName` STRING,
  `DestinationDisplay` STRING,
  `converted_expected` BIGINT,
  `converted_expected_arrival` BIGINT
  )
  COMMENT 'Transilien data passages'
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  STORED AS TEXTFILE
  LOCATION '/user/root/passages';
```

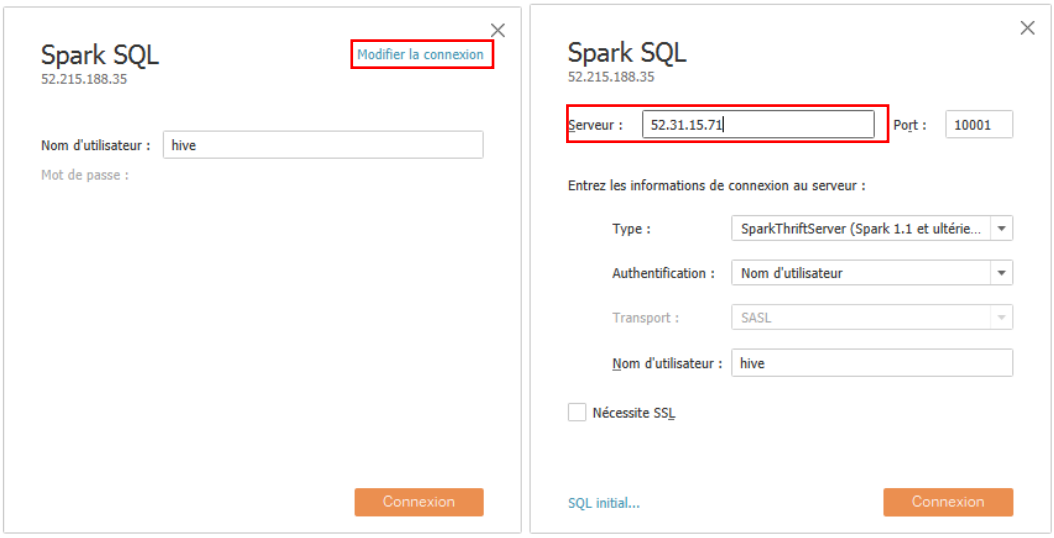
```
#Création d'une tâche planifiée
crontab -e

> */1 * * * * /usr/hdp/current/spark2-thriftserver/bin/beeline \
>       -u jdbc:hive2://localhost:10001 \
>       --outputformat=csv2 \
>       -e "INSERT OVERWRITE TABLE passages SELECT * FROM passage;"
```

9) Exploitation des données avec Tableau Software

L'exploitation des données est faite avec Tableau Software. Les données sont récupérées par l'intermédiaire de Spark Thrift Server qui permet d'exposer les données depuis un serveur SPark SQL.

Avant de lancement le tableau de bord, il est nécessaire de s'assurer du lancement du *producer* et du *consumer*. A l'ouverture du *dashboard*, il faut modifier la source de donnée Spark SQL et y entrer l'adresse IP publique IPv4 de l'instance EC2 Sandox-HDP.



a) Suivre des statistiques clefs de la ligne H

Les données sont exploitées dans le *dashboard* de Tableau Software dashboard_ligneH.twb. Ci-dessous un extrait du tableau de bord qui permet de :

- Déterminer le temps moyen d'attente sur la ligne station par station sur la dernière heure
- Déterminer le temps moyen d'attente globale sur la ligne sur la dernière heure
- Trier les stations par temps d'attente moyen sur la dernière heure
- Trouver la station avec le temps d'attente le plus élevée sur la dernière heure
- Trouver la station avec le temps d'attente le moins élevée sur la dernière heure

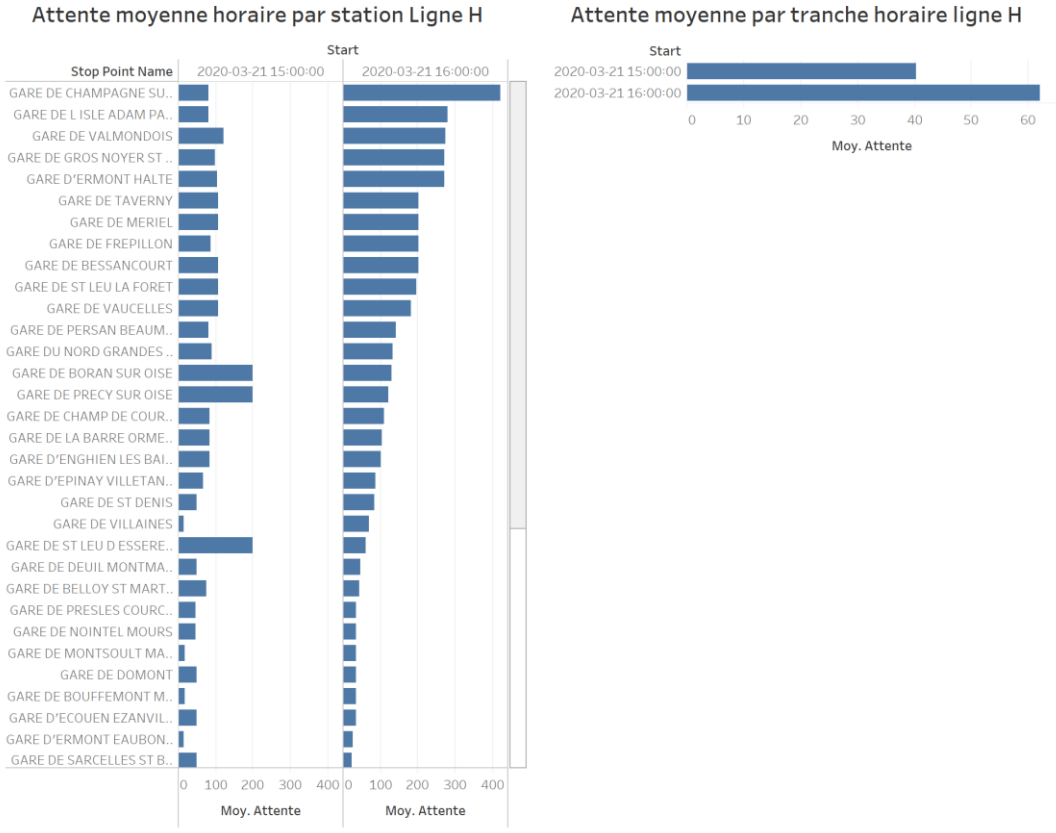


Figure 12: Tableau de bord - Temps d'attente

b) Estimer la position d'un train

La position du train est donnée dans le *dashboard* de Tableau Software progression_passage.twb.

Figure 13: Tableau de bord - Position train