Université du Québec à Chicoutimi

TRAVAIL 1
Tutoriel de technologie émergente

présenté à
Fabio Petrillo

dans le cadre du cours
8INF853 - Architecture des applications d'entreprise
du trimestre hiver 2020

réalisé par
Quentin Letort- LETQ12039703
Guillaume Routhier – ROUG10088306

TRAVAIL 1

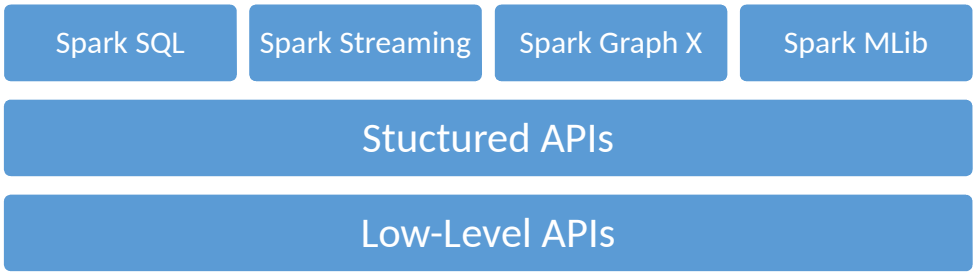
Le 23 février 2020

Contents

Présentation de Spark

Apache Spark est un framework de calcul distribué. Un ensemble de composants logiciels structurés développé à l'université de Californie à Berkeley. Spark est aujourd'hui un projet de la fondation Apache. C'est essentiellement un cadre applicatif de traitements « big data » pour effectuer des analyses complexes à grande échelle. Le cadre permet d'optimiser l'utilisation d'un « cluster » de ressources et ainsi minimiser le temps requis pour obtenir les résultats.

Spark est composé de plusieurs composants et librairies :



Nous reviendrons plus tard sur ces différents points en détail.

Contexte

Pourquoi avons-nous besoin d'un nouveau moteur de calculs et d'un modèle de programmation distribué pour l'analyse de données ? Tels que plusieurs autres changements dans le monde celui-ci proviennent en partie de raison économique liée aux applications ainsi qu'au matériel informatique.

Historiquement les processeurs des ordinateurs devenaient de plus en plus rapides d'année en année. Ainsi par défaut les applications construire sur ceux-ci devenait également plus rapide. Ceci à mener a l'établissement d'un large écosystème d'application conçu pour être exécuté principalement sur un seul processor.

Malheureusement cette tendance prit fin vers 2005 en raison de limitation au niveau de la dissipation de chaleurs lier à l'augmentation des cadences des processeurs. À partir de ce moment, les manufacturiers ont plutôt choisi d'ajouter plusieurs cœurs au processeur. Ceci a pour effets de modifier les patterns de création d'application pour utiliser un modèle à plusieurs processus.

Également, a ce même moment les technologies do stockage de données n'ont pas subi la même pression que les processeurs. Le coût de stockage est de plus en plus bas et les technologies d'acquisition de données sont également peu dispendieuses (camera, capteur, IOT, etc.). Ceci a donc causé une explosion des données disponibles pour le calcul.

Le résultat final est que nous disposons maintenant de quantité astronomique de données à analyser. Pour les analyser, nous avons donc besoin de grande plateforme de calculs parallèles distribués.

Historique

- 2009, Conception de Spark par Matei Zaharia lors de son doctorat a l'université de Californie à Berkeley. À l'origine la solution a pour but d'accélérer le traitement des systèmes Hadoop.
- 2013, Transmission de Spark a la fondation Apache. Il devient alors l'un des projets les plus actifs de la fondation.
- 2014, Spark a gagné le Daytona GraySort Contest dont l'objectif est de trier 100 To de donnés le plus rapidement possible. Ce record était préalablement détenu par Hadoop. Pour ce faire, Spark a utilisé 206 machines obtenant un temps d'exécution final de 23 minutes alors que Hadoop avait lui utilisé 2100 machines pour un temps d'exécution final de 72 minutes. La puissance de Spark fut démontrée en étant 3 fois plus rapide et en utilisant approximativement 10 fois moins de machines.[1]

Apache Spark philosophie

La philosophie de base pour Apache Spark est de fournir un système de calculs unifier et une collection de librairies pour le « big data »

Unifié

Le but est de fournir une plateforme pour la création d'application big data. Spark est désigné pour supporter une grande variété de tâches lier à l'analyse de donnée en partant du simple import de données de SQL aux tâches d'apprentissage machine ou de calculs de flux (streaming computation). Toutes ces tâches sont réalisées à partir du même moteur de calculs avec un ensemble d'API consistant. Ces API constants ont également comme fonction l'optimisation des calculs. Ainsi si vous fait une importation de données via SQL et ensuite vous demander l'exécution d'un calcul d'apprentissage machine sur le même jeu de données. Le moteur de Spark va combiner les 2 opérations et tenter d'optimiser le plus possible les étapes pour fournir l'exécution la plus performante possible.

Spark est créé en Scala, il supporte également Java, Python, SQL et R. certaines distinctions existent en fonction des langages utiliser, mais l'essentiel est disponible pour ceux-ci. Puisqu'il fut créé en Scala, le Scala et le Java sont les 2 langages les plus puissants pour utiliser Spark puisqu'il permet d'atteindre les API de bas niveau plus efficacement.

Moteur de calcul

Tout en tentant de rester unifié, Spark limite le plus possible l'étendue de son moteur de calcul. Par ceci nous désirons pointer que Spark prend en change l'import des données et le calcul en mémoire, il ne prend par contre pas en charge le stockage permanent des données en fin de calculs. Spark est par contre compatible avec une grande variété de produits de stockage tel que :

- Azure Storage
- Amazon S3
- Distributed File System :
    - Apache Hadoop
- Key-Value stores :
    - Apache Cassandra
- Message Buses
    - Apache Kafka

[1] https://fr.wikipedia.org/wiki/Apache_Spark

Spark ne fait que les utiliser, il ne stocke rien pour ces calculs et ne préfère aucune solution en particulier. L'idée derrière ceci est que les données utilisées pour les calculs existent déjà à travers une multitude de solutions de stockage. Les données sont coûteuses a bougé, ainsi Spark focus a exécuté les calculs sur les données peut importe leurs emplacements. Ce point est en fait ce qui le distingue des plateformes précédentes telles qu'apache Hadoop. Cette plateforme offrait le stockage (Hadoop file system), le moteur de calculs (MapReduce) et le service de cluster. Ce design limitait ainsi les calculs sur des données stockées ailleurs.

Librairies

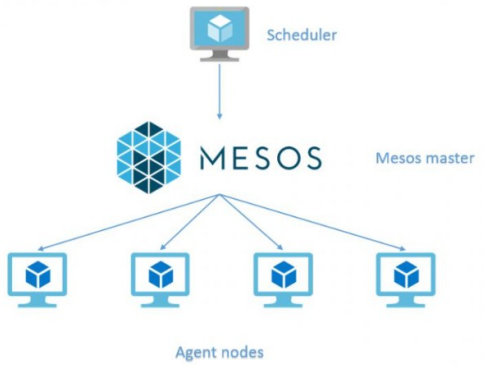
Le dernier composant étant les librairies qui ont été créer de façon unifier pour tirer profit du moteur de calcul. Spark supporte les librairies livrées avec le produit ainsi que celle créée par le public. Les librairies sont en quelques formes l'un des aspects les plus importants du projet et fournissent de plus en plus de fonctionnalité à Spark. Présentement les librairies offrent le support pour :

- Spark SQL
- Apprentissage machine (MLib)
- Stream processing
- Analyse de Graph (GraphX)
- Des centaines d'autres connecteurs et librairies …

Architecture

L'architecture de Spark est bien sûr de type distribuer. Ainsi Spark comprendre une node « driver » qui dirige les opérations, ainsi que plusieurs node « worker » qui exécute le travail envoyer. Spark peut-être déployé en plusieurs types de Clusters :

- Stand Alone (Mode interne de Spark, ne comprend pas de node pour « manager » le cluster.
- Apache Mesos

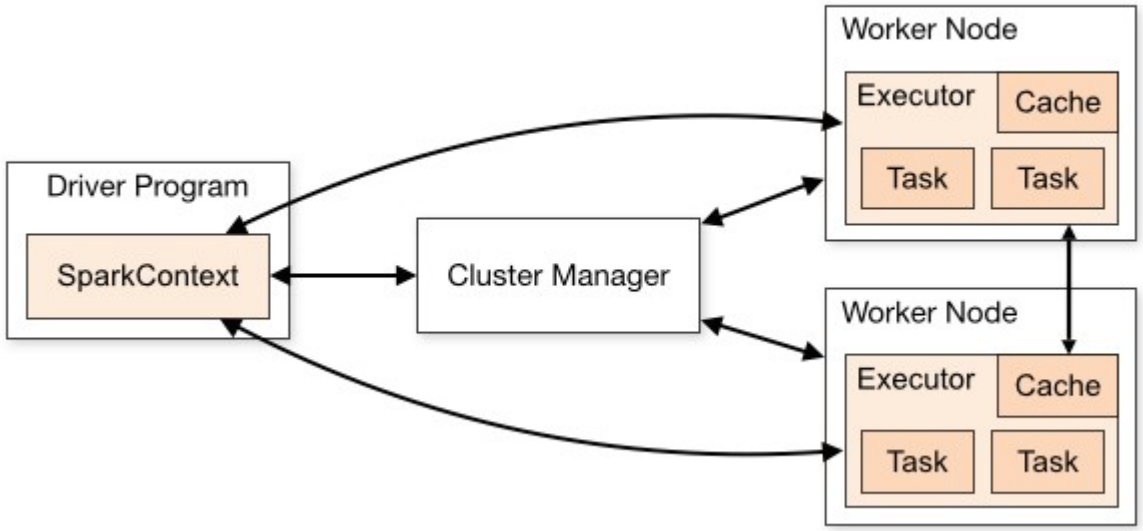


- Hadoop YARN

- Kubernetes

Il est également disponible en modèle SaaS via Azure ou AWS. Ceux-ci utilisent YARN pour l'hébergement du cluster et fournissent une panoplie d'autre utilitaire tels que Zepelin Notebook. La principalement différence entre les différents modèles est la maintenance et le déploiement des Clusters.

Le modèle de base est capable de supporter la scallabilité ainsi que l'échec des clusters. Advenant qu'une node tombe en problème, Spark va automatiquement renvoyé les jobs sur les autres nodes disponibles. Le mode stand alone par contre ne sera pas en mesure de réparer la node en problème comparativement aux autres modèles plus avancer qui eux peuvent corriger le problème et même déployé les nodes selon la demandes.



Outils et librairies

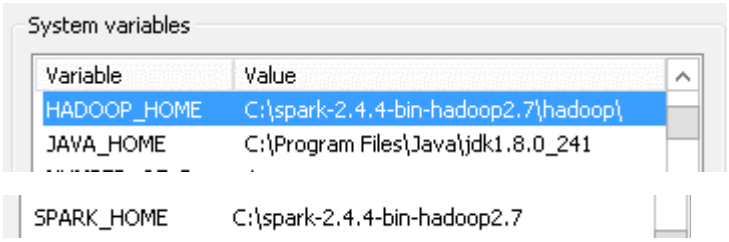
Tutoriel

Le tutoriel sera en 2 phases. Dans la première phase, nous démontrerons comment installer Spark via un cluster « stand alone » ainsi que comment produire 2 opérations via Jupyter-Notebook. Ensuite nous démontrerons les mêmes fonctionnalités via un cluster local pour que tous puissent essayer le tutoriel sans avoir besoin d'avoir un cluster d'ordinateur de disponible.

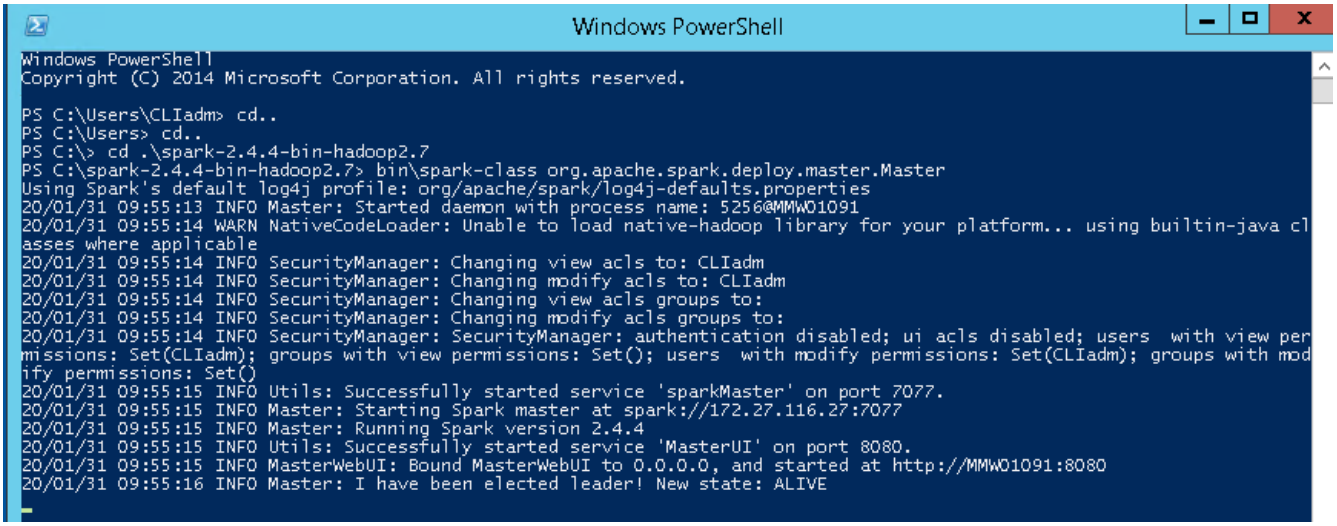
Installation du Cluster :

Installation du « driver » :

- Installation de Python 3.7
    - Configurer la variable d'environnement (automatique à l'installation de Python)
- Installation de Java JDK 8u241 (windows x64)
    - Configurer la variable d'environnement JAVA_HOME = <Path_to_jdk>
    - Ajouter JAVA_HOME au PATH
- Installation de Spark (2.4.4-bin-hadoop2.7)
    - Configurer la variable d'environement SPARK_HOME = <Path_to_Spark>
    - Ajouter SPARK_HOME au PATH
- Installation de winutils.exe de : https://github.com/steveloughran/winutils
    - Prendre version (hadoop-2.7.1)
    - Copier le exe dans SPARK_HOME\hadoop\bin
    - Configurer la variable d'environment HADOOP_HOME = %SPARK_HOME%\hadoop

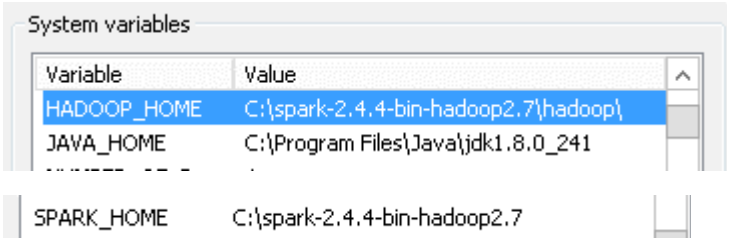


Pour tester l'installation allez dans SPARK_HOME et lancer: bin\spark-class org.apache.spark.deploy.master.Master



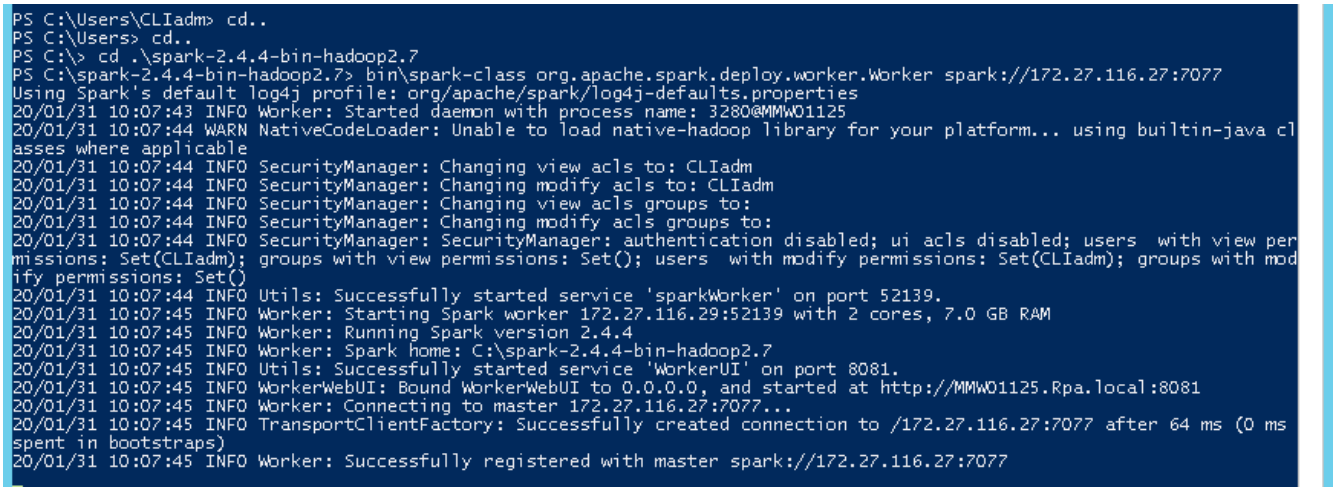
Installation des « worker node » :

- Installation de Python 3.7
    - Configurer la variable d'environnement (automatique à l'installation de Python)
- Installation de Java JDK 8u241 (windows x64)
    - Configurer la variable d'environnement JAVA_HOME = <Path_to_jdk>
    - Ajouter JAVA_HOME au PATH
- Installation de Spark (2.4.4-bin-hadoop2.7)
    - Configurer la variable d'environement SPARK_HOME = <Path_to_Spark>
    - Ajouter SPARK_HOME au PATH
- Installation de winutils.exe de : https://github.com/steveloughran/winutils
    - Prendre version (hadoop-2.7.1)
    - Copier le exe dans SPARK_HOME\hadoop\bin
    - Configurer la variable d'environment HADOOP_HOME = %SPARK_HOME%\hadoop
- Copier



Pour tester l'installation allez dans SPARK_HOME et lancer: bin\spark-class org.apache.spark.deploy.worker.Worker spark://<master_IP>:<port>

Le IP et le Port sont disponibles dans l'écran lors de lancement du Master. Dans notre exemple nous avons `spark://172.27.116.27:7077`



Faire la même opération sur chacun des workers.

Installation du poste de développement, ou version non-cluster :

Puisque dans cette démonstration le cluster et installer dans une zone sécurité, nous avons besoin de créer un poste de développement dans le même VLAN, ainsi notre ordinateur de développement sera configuré ainsi :

- Installez et configurez tels qu'un « worker node », ensuite :
  - Installer Anaconda
  - Create environment Python 3.7
  - Pip install jupyter
  - Pip install findspark

Ceci nous configure un cluster Spark qui peut exécuter les tâches que nous lui envoyons. Certains points sont par contre a retenir. Les solutions plus avancer de cluster peuvent s'assurer de faire l'installation automatique des dépendances (« jar ») alors que le mode standalone est plus compliquer si utiliser avec PySpark. Dans le cas qui nous intéresse nous devons ajouter manuellement les jar file sur toutes les machines du cluster.

- Sqljdbc42.jar dans SPARK_HOME\jars\

Également, la configuration du cluster sera utilisée pour la démonstration avec l'utilisation d'un serveur SQL en back end. Pour le tutoriel il sera plus simple de continuer avec une installation locale et un parquet file comme source de données. Ainsi 2 versions de code sera fournie, celle qui peut être exécuté sur le cluster et celle faite localement.

Pour la version locale, les fichiers AJOUTER ICI LES FICHIER DE LOG doit être extrait et sera la base d'entré.

Exemple simple d'utilisation des librairies de machine Learning

Utilisation avec cluster :

Example 1 : Clustering avec 3 features

Lecture des informations requises dans SQL et création du DataFrame :

In [1]:
```
import sys

import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("spark://172.27.116.27:7077") \
    .appName("RPA_Test") \
    .getOrCreate()
```

In [2]:
```
database = "RobotWorkForce"
table = "(SELECT [workNumber],[templateID],[robotID],[receivedTime],[startTime],[endTime],isnull([MasterType].[resolutic
user = "XXXX"
password = "XXXX"

DF = spark.read.format("jdbc") \
    .option("url", f"jdbc:sqlserver://172.27.116.14:1433;databaseName={database};") \
    .option("dbtable",table) \
    .option("user",user) \
    .option("password", password) \
    .option("driver","com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .load()

table2 = "(SELECT [itemKey] as 'workNumber',[templateID],[RobotStatus].[id] as 'robotID',[scheduledTime] as 'receivedTir
sDF2 = spark.read.format("jdbc") \
    .option("url", f"jdbc:sqlserver://172.27.116.14:1433;databaseName={database};") \
    .option("dbtable",table2) \
    .option("user",user) \
    .option("password", password) \
    .option("driver","com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .load()

#join des 2 tables
MasterLog = DF.unionAll(sDF2)
```

In [3]:
```
#enregistratement du dataframe dans spark pour l'utiliser via les commandes SQL
MasterLog.createOrReplaceTempView("MasterLog")
```

Ajouter d'un identifiant unique et split du Database pour entraînement et test :

In [4]:
```
#ajouter d'un Identifiant unique
from pyspark.sql.functions import *
df = spark.sql("SELECT workNumber, transactionTime, robotID, case when status = '' THEN 'unknown' ELSE status END as sta
df = df.withColumn("uniqueID",monotonically_increasing_id())

#retrait des valeurs null
df.na.drop()
```

Out[4]:
```
DataFrame[workNumber: string, transactionTime: int, robotID: int, status: string, uniqueID: bigint]
```

In [5]:
```
#division du dataframe en traning et test
seed = 0
train, test = df.randomSplit([0.7,0.3],seed)
```

Création du Pipeline de travail :

In [6]:
```
#Sting Indexer for Status
from pyspark.ml.feature import StringIndexer

stringIndexer1 = StringIndexer(inputCol="status",outputCol="statusIndex")
stringIndexer2 = StringIndexer(inputCol="robotID",outputCol="robotIndex")
```

In [7]:
```
#one Hot Encode Index
from pyspark.ml.feature import OneHotEncoder

oneHotEncoder1 = OneHotEncoder().setInputCol("statusIndex").setOutputCol("statusOneHot")
oneHotEncoder2 = OneHotEncoder().setInputCol("robotIndex").setOutputCol("robotOneHot")
```

In [8]:
```
#standard scaller
from pyspark.ml.feature import StandardScaler

standardScaler = StandardScaler(inputCol="features",outputCol="featuresScaled")
```

In [9]:
```
#Vector Assembler
from pyspark.ml.feature import VectorAssembler

vecAssembler = VectorAssembler(inputCols=['statusOneHot','robotOneHot','transactionTime'], outputCol="features")
```

In [10]:
```
#K-means
from pyspark.ml.clustering import KMeans

#use 6 Cluster
k = 6

kmeans = KMeans().setK(k).setSeed(1).setFeaturesCol("features")
```

In [11]:
```
#Pipeline Creation
from pyspark.ml import Pipeline
stages = [stringIndexer1,stringIndexer2,oneHotEncoder1,oneHotEncoder2,vecAssembler,kmeans]
pipeline = Pipeline().setStages(stages)
```

Entraînement du modèle et agrégation des résultats :

In [12]:
```
#Fitting the models using Parameters tuning, evaluation model and training validation split
kMeans_model = pipeline.fit(train)
```

In [13]:
```
#tranforming the models
transformed = kMeans_model.transform(df)
```

In [14]:
```
transformed = kMeans_model.transform(df).select('uniqueID', 'prediction','statusIndex')
rows = transformed.collect()
print(rows[:3])
```

```
[Row(uniqueID=0, prediction=0, statusIndex=0.0), Row(uniqueID=1, prediction=0, statusIndex=0.0), Row(uniqueID=2, pred
iction=0, statusIndex=0.0)]
```

Joint entre la table initiale et la table de résultat pour afficher tous les champs, enregistrement des résultats dans une table SQL pour faire une requête dans Spark, et affichage des groups de cluster trouver :

In [15]:
```
#join initial and final table on unique ID
df_pred = spark.createDataFrame(rows)

ta = df.alias('ta')
tb = df_pred.alias('tb')

join = ta.join(tb,ta.uniqueID == tb.uniqueID,how='left')
join.createOrReplaceTempView("MasterLog_Processed")
join.show(10)
```

```
+----------+---------------+-------+--------+--------+--------+----------+-----------+
|workNumber|transactionTime|robotID|  status|uniqueID|uniqueID|prediction|statusIndex|
+----------+---------------+-------+--------+--------+--------+----------+-----------+
|0000004570|             80|      4|Complete|      26|      26|         4|        0.0|
|0000005713|             42|      4|Complete|      29|      29|         0|        0.0|
|0000021257|             66|      9|Complete|     474|     474|         5|        0.0|
|0000041981|             89|     12|Complete|     964|     964|         0|        0.0|
|0000052709|             71|      6|Complete|    1677|    1677|         5|        0.0|
|0000052908|             84|     12|Complete|    1697|    1697|         0|        0.0|
|0000054517|             72|      6|Complete|    1806|    1806|         5|        0.0|
|0000056771|             37|      4|Complete|    1950|    1950|         5|        0.0|
|     34137|             74|     10|Complete|    2040|    2040|         0|        0.0|
|0000001740|             65|      7|Complete|    2214|    2214|         0|        0.0|
+----------+---------------+-------+--------+--------+--------+----------+-----------+
only showing top 10 rows
```

In [16]:
```
#group per Cluster
df = spark.sql("SELECT count(*), avg(transactionTime), prediction FROM MasterLog_Processed GROUP BY prediction ORDER BY
df.show()
```

```
+--------+--------------------+----------+
|count(1)|avg(transactionTime)|prediction|
+--------+--------------------+----------+
|   94154|    84.9118677910657|         0|
|    1165|  119.03090128755365|         1|
|     354|  113.67514124293785|         2|
|    4735|  115.44772967265048|         3|
|   16378|  110.57491757235316|         4|
|   53414|   100.0188340135545|         5|
+--------+--------------------+----------+
```

Enregistrement des résultats dans le SQL back-end :

In [17]:
```
#Output result to database
#set variable to be used to connect the database
database = "RobotWorkForce"
table = "dbo.Spark"
user = "XXXXX"
password  = "XXXXX"

#write the dataframe into a sql table
df.write.mode("overwrite") \
    .format("jdbc") \
    .option("url", f"jdbc:sqlserver://172.27.116.9:1433;databaseName={database};") \
    .option("dbtable",table) \
    .option("user",user) \
    .option("password", password) \
    .option("driver","com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

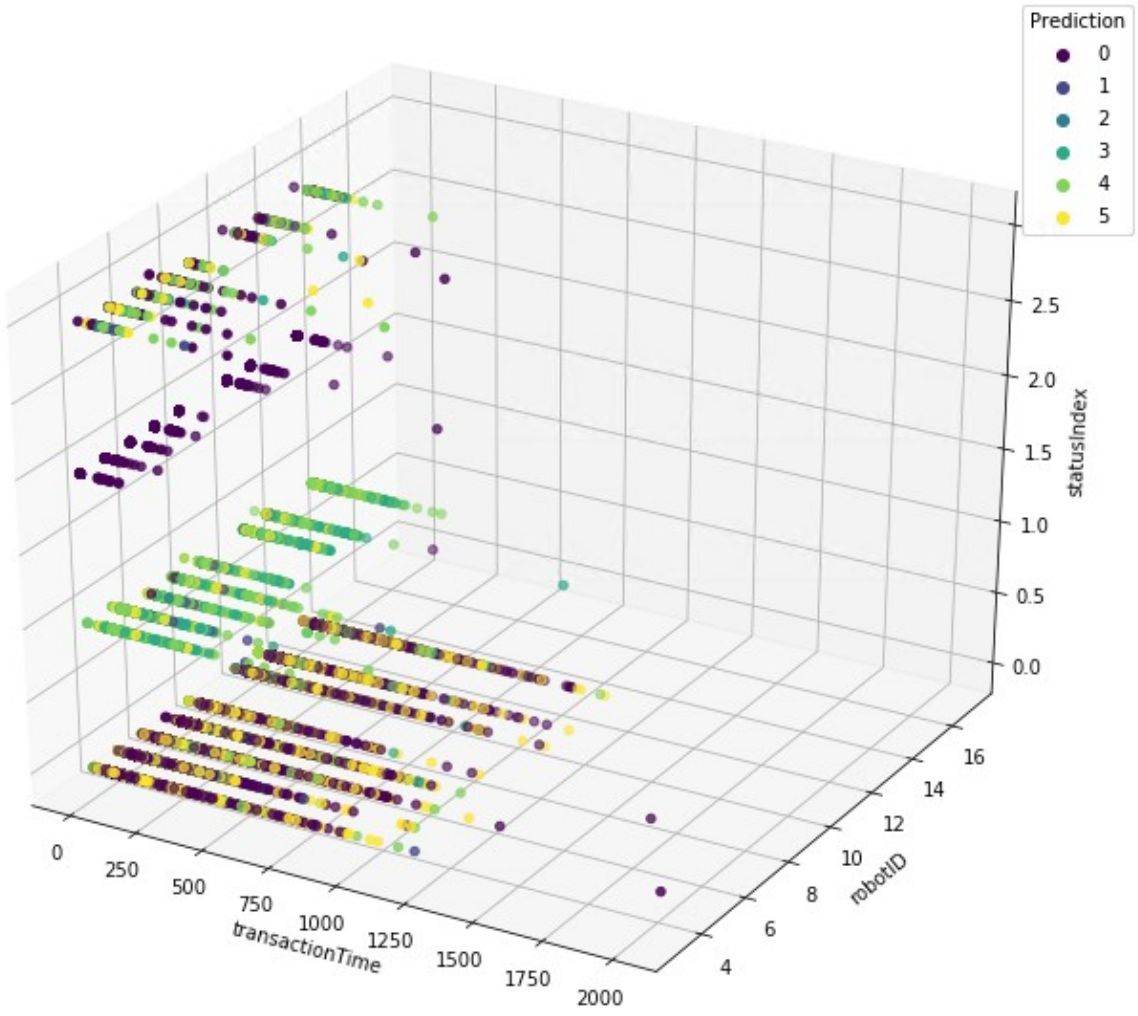
Affichage du résultat :

In [18]:
```
#Final display
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#group per Cluster
df = spark.sql("SELECT workNumber, transactionTime, robotID, statusIndex, prediction FROM MasterLog_Processed")

pddf_pred = df.toPandas().set_index('workNumber')
pddf_pred.head()

ax = plt.figure(figsize=(12,10)).gca(projection='3d')
scatter = ax.scatter(pddf_pred.transactionTime, pddf_pred.robotID, pddf_pred.statusIndex, c=pddf_pred.prediction)
ax.set_xlabel('transactionTime')
ax.set_ylabel('robotID')
ax.set_zlabel('statusIndex')
legend1 = ax.legend(*scatter.legend_elements(),loc="upper right",title="Prediction")
ax.add_artist(legend1)
plt.show()
```



Example 2 : Logistic Regression

Acquisition des données

In [1]:
```
import findspark
findspark.init()

import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("spark://172.27.116.27:7077") \
    .appName("RPA_Test") \
    .getOrCreate()
```

In [2]:
```
spark
```

Out[2]:

SparkSession - in-memory

SparkContext

Spark UI

Version

`v2.4.4`

Master

`spark://172.27.116.27:7077`

AppName

`RPA_Test`

In [3]:
```
database = "RobotWorkForce"
table = "(SELECT distinct ML.[ID],ML.[workNumber],ML.[templateID],ML.[robotID],ML.[receivedTime],ML.[startTime],ML.[endT
user = "xxxx"
password = "xxxx"

DF = spark.read.format("jdbc") \
    .option("url", f"jdbc:sqlserver://172.27.116.14:1433;databaseName={database};") \
    .option("dbtable",table) \
    .option("user",user) \
    .option("password", password) \
    .option("driver","com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .load()
```

Sauvegarde des données dans SQL Spark et recherche des données désirées

In [4]:
```
DF.createOrReplaceTempView("MasterLogData")
DF.printSchema()
```

```
root
 |-- ID: integer (nullable = true)
 |-- workNumber: string (nullable = true)
 |-- templateID: integer (nullable = true)
 |-- robotID: integer (nullable = true)
 |-- receivedTime: timestamp (nullable = true)
 |-- startTime: timestamp (nullable = true)
 |-- endTime: timestamp (nullable = true)
 |-- ResolutionTimeObjective: double (nullable = true)
 |-- transactionTime: integer (nullable = true)
 |-- status: string (nullable = true)
 |-- data: string (nullable = true)
```

In [5]:
```
#Get Data
from pyspark.sql.functions import *
df = spark.sql("SELECT id, workNumber, case when status = '' THEN 'unknown' ELSE status END as status, replace(replace(s
```

In [6]:
```
#split set training and testing
seed = 0  # set seed for reproducibility

train, test = df.randomSplit([0.7,0.3],seed)
```

Création du pipeline

In [7]:
```
from pyspark.ml.feature import Tokenizer, CountVectorizer, IDF, StringIndexer,HashingTF

#String Indexer
stringIndexer = StringIndexer(inputCol="status",outputCol="label").setHandleInvalid("skip")

#Tokenizer
tokenizer = Tokenizer(inputCol="data",outputCol="Token")

#Vectorizer
vectorizer = CountVectorizer(inputCol="Token", outputCol="rawFeatures")

#TD-IDF
tf = HashingTF() \
    .setInputCol("Token") \
    .setOutputCol("TFout") \
    .setNumFeatures(10000)

idf = IDF() \
    .setInputCol("TFout") \
    .setOutputCol("IDFOut")

#Vector Assembler
from pyspark.ml.feature import VectorAssembler
FEATURES_COL = ['rawFeatures','IDFOut']
vecAssembler = VectorAssembler(inputCols=FEATURES_COL, outputCol="features")
```

In [8]:
```
#Logistic Regression Classifier
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")
```

In [9]:
```
#Pipeline Creation
from pyspark.ml import Pipeline
stages = [stringIndexer,tokenizer,vectorizer,tf,idf,vecAssembler,lr]
pipeline = Pipeline().setStages(stages)
```

Création d’un ParamGrid, Model d’évaluation et Training Split

In [10]:
```
#Param Tuning
from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder() \
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
    .addGrid(lr.regParam, [0.1, 2]) \
    .build()
```

In [11]:
```
#Evaluation Model
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator() \
    .setMetricName("areaUnderROC") \
    .setRawPredictionCol("prediction") \
    .setLabelCol("label")
```

In [12]:
```
#Training validation split
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit() \
    .setTrainRatio(0.75) \
    .setEstimatorParamMaps(params) \
    .setEstimator(pipeline) \
    .setEvaluator(evaluator)
```

In [*]:
```
#Fitting the models using Parameters tuning, evaluation model and training validation split
tvsFitted = tvs.fit(train)
```

Validation du modèle, et affichage des résultats

```
In [14]: #tranforming the models
transformed = tvsFitted.transform(test)

In [15]: #Brut result
evaluator.evaluate(transformed)

Out[15]: 0.7557601773715992

In [16]: #Create Result table and output table
transformed.createOrReplaceTempView("Result")
finalDF = spark.sql("SELECT status,label, prediction, count(*) FROM Result GROUP BY status, label,prediction ORDER BY st
finalDF.show()
```

```
+-----------------+-----+----------+--------+
|           status|label|prediction|count(1)|
+-----------------+-----+----------+--------+
|         Complete|  0.0|       0.0|  248168|
|         Complete|  0.0|       1.0|    2082|
|         Complete|  0.0|       2.0|    1106|
|         Complete|  0.0|       3.0|       1|
|Complete-Reassign|  1.0|       0.0|    9030|
|Complete-Reassign|  1.0|       1.0|    5540|
|Complete-Reassign|  1.0|       2.0|     112|
|             Fail|  3.0|       0.0|    1606|
|             Fail|  3.0|       1.0|      56|
|             Fail|  3.0|       2.0|     185|
|          unknown|  2.0|       0.0|      90|
|          unknown|  2.0|       1.0|       2|
|          unknown|  2.0|       2.0|    5867|
+-----------------+-----+----------+--------+
```

```
In [17]: spark.stop()
```

Utilisation hors cluster (local) :

2 script ont été créer pour extraire les données des bases de données utiliser dans le modèle avec Cluster, ces bases de données sont ensuite sauvegardées « parquet » file pour être utilisé localement. Ceci sert à faire le tutoriel sans cluster et sans accès au BD SQL utiliser via le cluster. Ces scripts sont exécutés localement à l'extérieur du cluster, mais serve à démontrer comment se connecter dans un SQL et sauvegarder les données.

Spark – save to parquet / Spark – save to parquet (with data)

In [1]:
```
import findspark
findspark.init()
```

In [2]:
```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
spark
```

Out[2]:
SparkSession - in-memory
SparkContext

Spark UI
Version
`v2.4.4`
Master
`local[*]`
AppName
`pyspark-shell`

In [3]:
```
database = "RobotWorkForce"
table = "(SELECT distinct ML.[ID],ML.[workNumber],ML.[templateID],ML.[robotID],ML.[receivedTime],ML.[startTime],ML.[endT
user = "XXXXXX"
password = "XXXXXX"
```

In [4]:
```
DF = spark.read.format("jdbc") \
    .option("url", f"jdbc:sqlserver://172.27.116.14:1433;databaseName={database};") \
    .option("dbtable",table) \
    .option("user",user) \
    .option("password", password) \
    .option("driver","com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .load()
```

In [9]:
```
#write to parquet file
MasterLog.write.parquet("C:\\8INF853 - Architecture des applications d'entreprise\MasterLog")
```

Un coup ces scripts exécute vous posséder les “parquet file” et pouvez donc exécute les autres scripts localement.

Les scripts précédents peuvent maintenant être exécuté, il suffit de modifier le début pour ceci :

In [1]:
```
import findspark
findspark.init()
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
spark
```

Out[1]:
SparkSession - in-memory
SparkContext

Spark UI
Version
`v2.4.4`
Master
`local[*]`
AppName
`pyspark-shell`

In [2]:
```
MasterLog = spark.read.parquet("C:\\8INF853 - Architecture des applications d'entreprise\\MasterLogData")

#register the dataframe in the SQL server
MasterLog.createOrReplaceTempView("MasterLogData")

from pyspark.sql.functions import *
df = spark.sql("SELECT id, workNumber, status, replace(replace(replace(data,'<',' '),'>', ' '),'/',' ') as data FROM Mas
```

Référence

https://fr.wikipedia.org/wiki/Apache_Spark

Spark : The Definitive Guide. Bill Chambers & Matei Zaharia. O’Reilly Media 2008.