



SPARK API

(C)Hurence

Version v1.0, 18.02.2020: First draft

Table des matières

Introduction.....	1
Spark Shell.....	1
Projet java	1
les packages à importer	2
Les readers.....	3
Transformation des données.....	9
Les writers	11

Introduction

L'api-spark est une api qui sert à lire les données à l'aide de spark, les chunkifier et les injecter finalement dans Solr.

Il y a deux façons pour utiliser cette api :

Spark Shell

Pour installer apache spark vous pouvez télécharger cette archive : [spark-2.3.4-bin-without-hadoop.tgz](#)

Les commandes suivantes vous permettront de faire une installation locale.

```
# installer Apache Spark 2.3.4 and unpack it
cd $HDH_HOME
wget https://archive.apache.org/dist/spark/spark-2.3.4/spark-2.3.4-bin-without-hadoop.tgz
tar -xvf spark-2.3.4-bin-without-hadoop.tgz
rm spark-2.3.4-bin-without-hadoop.tgz


# add two additional jars to spark to handle our framework
wget -O spark-solr-3.6.6-shaded.jar
https://search.maven.org/remotecontent?filepath=com/lucidworks/spark/spark-solr/3.6.6/spark-solr-3.6.6-shaded.jar
mv spark-solr-3.6.6-shaded.jar $HDH_HOME/spark-2.3.4-bin-without-hadoop/jars/
cp $HDH_HOME/historian-1.3.4-SNAPSHOT/lib/loader-1.3.4-SNAPSHOT.jar $HDH_HOME/spark-2.3.4-bin-without-hadoop/jars/
```

Après avoir dézipper l'archive, vous devez vous mettre dans le dossier "bin", Lancer l'interpréteur et lancer la commande suivante :

```
spark-shell --jars $HDH_HOME/spark-2.3.4-bin-without-hadoop/jars/loader-1.3.4-SNAPSHOT.jar, $HDH_HOME/spark-2.3.4-bin-without-hadoop/jars/spark-solr-3.6.6-shaded.jar
```

Il s'agit d'une invite de commandes interactive permettant de communiquer directement avec un cluster Spark local sans oublier les jars qui contiennent les APIs de HDH.

Projet java

Il faut ajouter les JARs dans la CLASSPATH pour utiliser les APIs de data historian, ou vous pouvez ajouter les jars qui existent sur internet par l'ajout des dépendances dans le fichier pom.xml de Apache Maven par exemple.  un grand nombre de JARs sont disponibles sur les repositories Maven. Il faut se rendre au site <http://www.mvnrepository.com/>

les packages à importer

Pour utiliser cette API il faut importer les packages contenant les classes assurant son fonctionnement.

```
import com.hurence.historian.model.ChunkRecordV0
import com.hurence.historian.spark.ml.Chunkyfier
import com.hurence.historian.spark.sql
import com.hurence.historian.spark.sql.functions._
import com.hurence.historian.spark.sql.reader.{MeasuresReaderType, ReaderFactory}
import com.hurence.historian.spark.sql.writer.{WriterFactory, WriterType}
import org.apache.spark.sql.functions.typedLit
import com.lucidworks.spark.util.SolrSupport
import org.apache.commons.cli.{DefaultParser, Option, Options}
import org.apache.spark.sql.SparkSession
import org.slf4j.LoggerFactory
import org.apache.spark.sql.types._
```

Les readers

Il faut spécifier le type de reader qu'on doit utiliser et le type de données qu'on veut lire :

En tant que Mesure (la mesure est un point dans le temps avec une valeur *double* un nom et quelques *tags*) identifié sous cette forme :

```
case class Measure(name: String,
                  value: Double,
                  timestamp: Long,
                  year: Int,
                  month: Int,
                  day: String,
                  hour: Int,
                  tags: Map[String,String])
```

- **EVOA_CSV** : Les données sont réparties sous cette forme : *name, value, quality, code_install, sensor, timestamp, time_ms, year, month, week, day*. Ces données seront ordonnées par le *name* et *time_ms*.

La classe *EvoaCSVMasuresReader* contient une méthode de configuration :

Table 1. EVOA_CSV configuration

option	type	description	valeurs possible	valeur par défaut
inferSchema	boolean	déduit automatiquement le schéma d'entrée à partir des données	true or false	false
sep	character	définit le caractère utilisé pour séparer les données	n'importe quel caractère	;
header	boolean	utilise la première ligne comme noms de colonnes	true ou false	false
dateFormat	String		Le format de la date	

On lit le fichier avec la méthode `spark.read` de `spark`. Les options sont :

Table 2. *spark.read* options (CSV)

option	type	description	valeurs possible	valeur par défaut
header	boolean	transforme la première ligne comme noms des colonnes	true ou false	false

option	type	description	valeurs possible	valeur par défaut
sep	character	définit le caractère utilisé pour séparer les données	n'importe quel caractère	,
quote	character	le caractère de citation	n'importe quel caractère	"
escape	character	utilisé pour échapper un caractère	n'importe quel caractère	\
parserLib	string	spark-csv parser	commons or univocity	commons
mode	string	mode d'analyse	<u>PERMISSIVE</u> : essayez d'analyser toutes les lignes: des valeurs nulles sont insérées pour les jetons manquants et les jetons supplémentaires sont ignorés. <u>DROPMALFORMED</u> : drop lignes qui ont moins ou plus de jetons que prévu ou jetons qui ne correspondent pas au schéma. <u>FAILFAST</u> : abandonner avec une RuntimeException si une ligne mal formée est rencontrée.	PERMISSIVE
charset	string	utilisé pour spécifier le codage des caractères	n'importe quel codage des caractères	UTF-8
inferSchema	boolean	déduit automatiquement le schéma d'entrée à partir des données	true or false	false
comment	character	utilisé pour sauter les lignes qui commencent par le caractère spécifié	n'importe quel caractère	#
nullValue	string	chaîne qui indique une valeur nulle, tous les champs correspondant à cette chaîne seront définis comme nuls dans le DataFrame	n'importe quelle chaîne de caractères	

option	type	description	valeurs possible	valeur par défaut
dateFormat	string	chaîne qui indique le format de date à utiliser lors de la lecture de dates ou timestamp	Les formats de date personnalisés suivent les formats de java.text.SimpleDateFormat	

- **ITDATA_CSV** : Les données sont réparties sous cette forme : *name, value, timestamp, year, month, day, hour, tags*. Ces données seront ordonnées par le *name* et *timestamp*.

la classe *ITDataCSVMeasuresReaderV0* contient une méthode de configuration :

Table 3. *ITDATA_CSV* configuration

option	type	description	valeurs possible	valeur par défaut
inferSchema	boolean	déduit automatiquement le schéma d'entrée à partir des données	true or false	false
delimiter	character	définit le caractère utilisé pour séparer les données	n'importe quel caractère	,
header	boolean	utilise la première ligne comme noms de colonnes	true or false	false
dateFormat	String	chaîne qui indique le format de date à utiliser lors de la lecture de dates ou timestamp	Les formats de date personnalisés suivent les formats de java.text.SimpleDateFormat	

- **PARQUET** : Le parquet est un format en colonnes pris en charge par de nombreux autres systèmes de traitement des données. Spark SQL prend en charge la lecture et l'écriture de fichiers Parquet qui préserve automatiquement le schéma des données d'origine. Veuillez consulter la documentation d'apache spark pour les fichiers parquet : [Parquet Files - Spark 3.0.0 Documentation](#)
- **GENERIC_CSV** : Les données sont réparties sous cette forme : *name, value, timestamp, year, month, day, hour, tags*.

Table 4. *GENERIC_CSV* configuration

option	type	description	valeurs possible	valeur par défaut
inferSchema	boolean	déduit automatiquement le schéma d'entrée à partir des données	true ou false	false
delimiter	character	définit le caractère utilisé pour séparer les données	n'importe quel caractère	,
header	boolean	utilise la première ligne comme noms de colonnes	true ou false	false
dateFormat	String	Le format de la date		name
timestampField	String			timestamp
timestampDateFormat	String	le format de la timestamp	s or ms	s
valueField	String			value
tagsFields	String			tag_a,tag_b

En tant que Chunk (un chunk est une ensemble de mesures continues dans un intervalle de temps regroupés par une date, nom et des *tags*) identifié sous cette forme :

```
case class Chunk(name: String,
                 day:String,
                 start: Long,
                 end: Long,
                 chunk: String,
                 count: Long,
                 avg: Double,
                 stddev: Double,
                 min: Double,
                 max: Double,
                 first: Double,
                 last: Double,
                 sax: String,
                 tags: Map[String,String])
```

- **PARQUET** : Le parquet est un format en colonnes pris en charge par de nombreux autres systèmes de traitement des données. Spark SQL prend en charge la lecture et l'écriture de fichiers Parquet qui préserve automatiquement le schéma des données d'origine. Veuillez consulter la documentation d'apache spark pour les fichiers parquet : [Parquet Files - Spark 3.0.0 Documentation](#)
- **SOLR** : On utilise le spark.read pour la lecture des données en spécifiant que le format est solr. En résultat, les données seront être répartis sous cette forme : day, start, end, count, avg, stddev,

min, max, first, last, sax, value, chunk, tags

EXEMPLE: Dans ce cas on spécifie le reader *MeasuresReaderType* et nos données sont en format *GENERIC_CSV* :

```
val reader = ReaderFactory.getMeasuresReader(MeasuresReaderType.GENERIC_CSV)
val measuresDS = reader.read(sql.Options(
  origpath,
  Map(
    "inferSchema" -> "true",
    "delimiter" -> ",",
    "header" -> "true",
    "nameField" -> "metric_name",
    "timestampField" -> "timestamp",
    "timestampDateFormat" -> "ms",
    "valueField" -> "value",
    "tagsFields" -> "metric_id,warn,crit"
  )))
```

Les données brutes :

metric_id	timestamp	value	metric_name	warn	crit	min	max
091c334c-a90a-4d8...	1575157723	13.375	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575157423	13.5	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575157123	13.375	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575156823	13.5	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575156523	13.75	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575156223	2.125	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575155923	2.25	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575155623	1.875	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575155323	8.5	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575155023	17.375	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575154723	15.375	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575154424	18.5	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575154124	18.75	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575153824	24.0	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575153524	18.125	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575153224	17.875	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575152924	14.375	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575152624	20.0	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575152324	12.0	cpu_prct_used	85.0	95.0	null	null
091c334c-a90a-4d8...	1575152025	14.25	cpu_prct_used	85.0	95.0	null	null

Les données après la transformation en une mesure :

Transformation des données

chunkyfier : une méthode qui transforme les mesures en chunks.

Table 5. Les paramètres du Chunkyfier et ses méthodes

paramètre	type	description	valeurs possibles	valeur par défaut
setValueCol	String	définit la colonne des valeur	un nom d'une colonne	"value"
setTimestampCol	String	définit la colonne des timestamps	un nom d'une colonne	"timestamp"
setChunkCol	String	définit la colonne des chunks encodés	un nom d'une colonne	"chunk"
doDropLists	boolean	sert a jeter le colonnes des valeurs et timestamps	true or false	true
setDateBucketFormat	String	définit le format de la date suivant les formats proposé par le langage java	Les formats de date personnalisés suivent les formats de java.text.SimpleDateFormat	"yyyy-MM-dd"
setGroupByCols	Array[String]	définit les noms des colonnes qu'on va faire le groupeBy avec	ensemble des noms des colonnes	
setSaxAlphabetSize	Integer	définit la valeur du paramètre saxAlphabetSize	[0 , 20]	5
setSaxStringLength	Integer	définit la valeur du paramètre saxStringLength	[0 , 10000]	20
setChunkMaxSize	Integer	définit la valeur du paramètre chunkMaxSize		1440
transform	Dataset[_]	transforme les mesures en chunks		
transformSchema	StructType	transforme le schéma des données		

EXEMPLE:

```
val chunkyfier = new Chunkyfier().setGroupByCols(Array( "name", "tags.metric_id"))
val chunksDS = chunkyfier.transform(measuresDS).as[ChunkRecordV0]
```

Les données avant la transformation :

Les writers

Il existe deux types de writers : un qui sert à écrire des données de type *parquet* et un pour écrire des données *solr*.

- **Parquet** : pour écrire les données en parquet il faut utiliser la méthode *write* de la classe *parquetChunksWriter* où on doit spécifier les options et les données à écrire.

Table 6. Les paramètres de la méthode *write* de *ParquetChunkWriter*

paramètre	type	description	valeurs possibles	valeur par défaut
option.config	Map[String, String]	définit la configuration		
partitionBy	String	Partitionner les données suivant une certaine periode		"day"
mode	String	définit le mode d'écriture	"append", "overwrite", "error", "errorifexists", "ignore"	"append"
path	String	définit le chemin de dossier de sauvegarde		

- **solr** : Pour injecter les données dans solr il faut utiliser la méthode *write* de la classe *solrChunksWriter* où on doit spécifier les options et les données à écrire.