****

Table des matières

[**1 - Introduction 4**](#_heading=h.gjdgxs)

[**2 - Contexte métier 5**](#_heading=h.30j0zll)

[**3 - Architecture technique 6**](#_heading=h.1fob9te)

[**3-1. Schéma d’architecture générale 6**](#_heading=h.3znysh7)

[**3-2. Flux évènement 6**](#_heading=h.2et92p0)

[3-2-1. Technologie 6](#_heading=h.tyjcwt)

[3-2-2. Implémentation 7](#_heading=h.3dy6vkm)

[**3-3. Flux référentiel 9**](#_heading=h.1t3h5sf)

[**3-4. Consolidation des données 12**](#_heading=h.4d34og8)

# Introduction

Ce document vise à présenter l’architecture technique proposée pour répondre aux besoins du projet de NoSQL DE2. Le projet est basé sur un problème métier réel que les ingénieurs d’Enedis ont dû résoudre lors du déploiement des compteurs Linky.

# Contexte métier

Le problème présenté est un problème standard dans le monde de l’IoT et du Big Data :

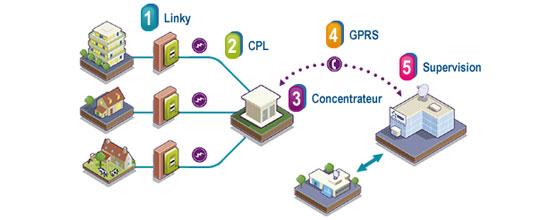


Figure : Le réseau des compteurs Linky

Un parc d’objets communiquant de grande ampleur est déployé sur un large territoire. Ces objets communicants remontent des données métiers spécifiques. Ces données doivent être ingérées et stockées de la manière la plus pertinente possible afin de faciliter leur exploitation. Ces données sont par la suite agrégées puis traitées afin d’en extraire des indicateurs pertinents.

Dans notre exemple, les compteurs Linky représentent les objets communicants. Les évènements représentent le flux de données métiers spécifiques. Elles sont remontées sur un cluster Kafka et stockées sur Cassandra. Enfin, ce document va présenter les étapes qui nous permettront d’agréger et traiter les données pour récupérer les indicateurs pertinents.

# Architecture technique

## Schéma d’architecture générale

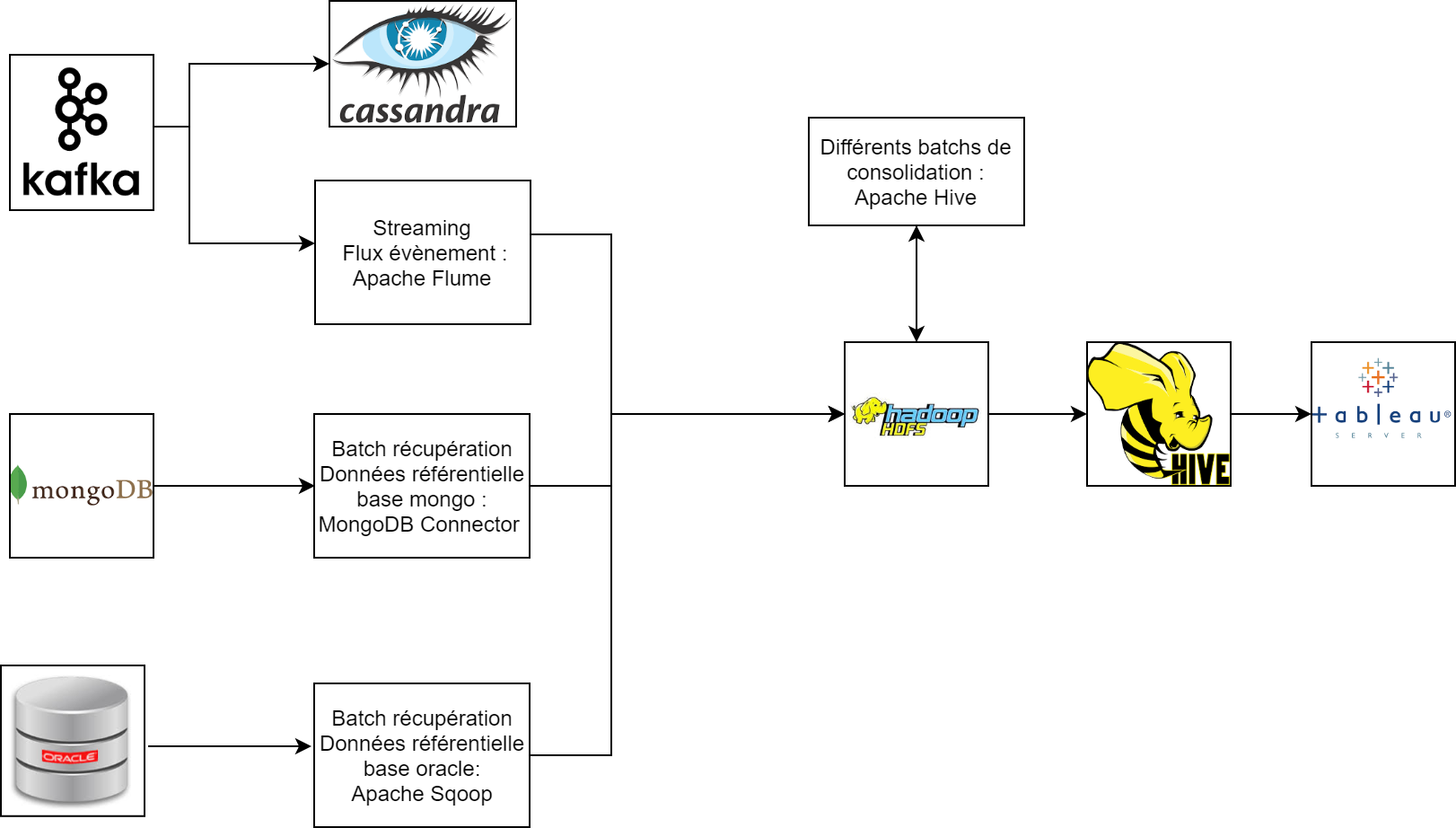


Figure : Schéma d’interaction des technologies proposées

Ce schéma décrit toutes les technologies utilisées ainsi que leurs interactions. Il permet aussi d’identifier la position des différents batch d’extraction et de consolidation des données. Nous allons maintenant détailler les différentes connexions entre les composants

## Flux évènement

### Technologie

Les données évènements arrivent dans le composant Kafka sous la forme d’un flux. L’architecture déjà en place stocke les données en temps réel dans une base Cassandra. Nous avons deux choix pour récupérer ces données :

1. Utiliser un connecteur entre Cassandra et HDFS et créer un batch d’exécution quotidien faisant une extraction des données arrivées lors de la précédente journée pour les stocker dans HDFS
2. Utiliser un connecteur entre Kafka et HDFS afin de recevoir les données provenant du flux d’évènement dans HDFS en temps réel

Nous avons décidé de privilégier la seconde option car elle présente plusieurs avantages. D’abord les données sont disponibles immédiatement dans HDFS et cela permettrait dans le cas où on souhaiterait des indicateurs plus temps réel d’avoir des données plus fraîches à disposition. Ensuite cela permet de lisser la charge de travail au cours de la journée. Enfin, il existe une solution adapté facile à mettre en œuvre pour réaliser ce travail : Apache Flume.

Apache Flume est un service distribué et à haute disponibilité permettant de collecter, agréger et déplacer de grandes quantités de données de logs. Le modèle utilisé par Flume est le suivant

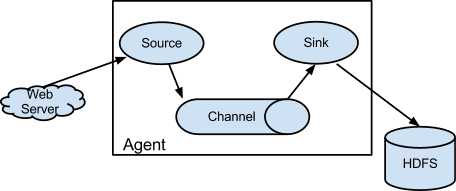


Figure : Modèle de données de Flume

Dans notre cas nous n’aurons pas besoin de réaliser de traitement sur les données, nous allons nous servir de Flume pour streamer les données de Kafka dans une table Hive. Ce qui est un cas d’utilisation courant pour la technologie

Ainsi, notre agent Flume ressemblera à cela :

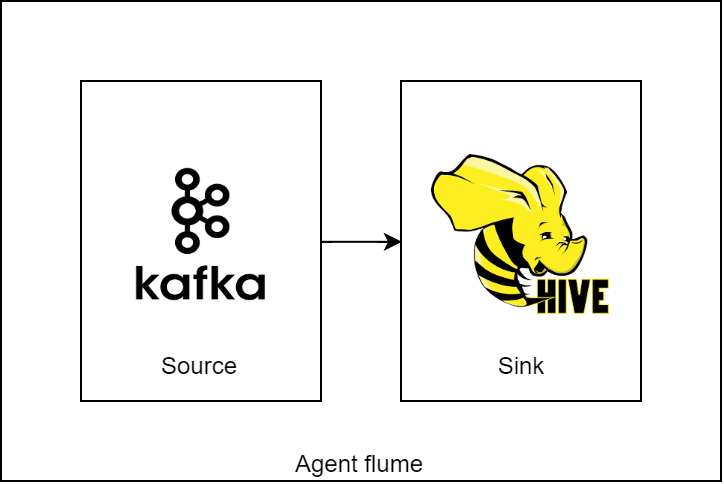


Figure : Structure de l’agent Flume

De plus il faudra prévoir une transformation de la date métier d’occurrence des évènement pour ajouter une donnée date\_event\_occur\_formatted au format ‘YYYY-MM-DD’ afin de faciliter le traitement.

Les données ne seront alors pas encore tout à fait exploitables. En effet les limitations de Hive Streaming nous poussent à stocker les données partitionnées par date technique et non par date d’occurrence. Nous aurons donc un batch supplémentaire permettant de transférer les données quotidiennement enregistrées par le stream et de les replacer dans une autre table partitionnée par date métier.

### Implémentation

Création d’une table Hive recevant les données évènements :

Le partitionnement de la table se fait avec une chaîne de caractère représentant la date d’occurrence de l’évènement. Elle est représentée sous la forme d’une seule chaîne au format ‘YYYY-MM-DD’.

D’une part le partitionnement par jour est pertinent car la plupart des indicateurs que nous allons calculer ont une granularité au niveau du jour. Si nous voulons récupérer un mois entier, il n’y a alors qu’à récupérer 31 partitions au maximum ce qui est raisonnable.

D’autre part, le format de la clé de partition va nous permettre de grandement simplifier les requêtes. On va pouvoir ainsi utiliser les commandes SQL LIKE, IN ou BETWEEN ce qui ne sont pas utilisable si on possède des partitions imbriquées.

Les limitations de Hive streaming nous forcent à stocker les données au format orc. Cette table va servir de table tampon partitionnée par date technique. Il va ensuite falloir implémenter un batch qui va venir transférer ces données dans une table Hive partitionnée par date métier, au format ‘YYYY-MM-DD’.

Les différents scripts à exécuter sont les suivants :

|  |  |
| --- | --- |
| Script | Localisation dans le repo git |
| Création de la table de réception des données streamées | /Deploiement/EventData/eventDataStreaming.hql |
| Création de la table des données partitionnées par date métier | /Deploiement /EventData/eventData.hql |
| Configuration de l’agent Apache Flume | /Deploiement /EventData/flume\_agent.conf |
| Création de l’agent Apache Flume | /Deploiement /EventData/run.sh |
| Transfert des données et partitionnement par date métier | /Deploiement /EventData/extractData.hql |

Afin de faire tourner tout cela il va falloir planifier un batch d’exécution qui va venir exécuter le script exctractData.hql régulièrement.

Ce script nous permet de déplacer les données de la table de streaming vers la table où l’on va construire nos requêtes. Il nécessite deux paramètres : La date technique source dont on va récupérer la partition et la date métier cible dont on va stocker les données. Une fois que les données sont transférées on les supprime de la table source. Ainsi on peut répartir la charge de travail tout au long de la journée en transférant les données toutes les heures.

Il faudra aussi créer un script qui va s’exécuter quotidiennement pour aller chercher les quelques données qui sont en retard de moins de 30 jours. On considère qu’au-delà de trente jours les données qui ne sont pas arrivées sont perdues, leur quantité étant négligeable.

Toutes ces étapes nous permettent d’avoir une table Hive mise à jour quotidiennement de la structure suivante

|  |  |  |
| --- | --- | --- |
| Nom colonne | Type | Description |
| id\_evt | STRING | id technique de l’événement |
| date\_system\_evt | STRING | date technique d’insertion de l’événement dans le SI |
| date\_occur\_evt | STRING | date métier d’occurrence de l’événement |
| date\_occur\_evt\_formatted | STRING | date métier d’occurrence de l’événement, format YYYY-MM-DD |
| type\_evt | STRING | type de l’événement |
| type\_equipement | STRING | type de l’équipement |
| id\_equipement | STRING | id métier de l’équipement à l’origine de l’événement |
| infos\_div | STRING | texte libre |
| DateStamp | STRING | Clé de partition au format ‘YYYY-MM-DD’ |

## Flux référentiel

**3-3-1. Technologie**

Les données référentielles sont stockées dans des bases :

* MongoDB pour les tables de déploiement.
* Oracle pour les tables de réseau.

Pour pouvoir récupérer toutes ces données correctement, plusieurs solutions s’offraient à nous :

* Pour MongoDB :
  + Utiliser un connecteur Hadoop à intégrer, permettant ainsi d’avoir MongoDB comme source d’entrée de données.
  + Utiliser des batch d’exécution quotidien en Spark ou en Hive, pour ainsi transférer les données dans HDFS.

Nous avons décidé de partir sur la première solution, car elle est celle qui est la plus simple à mettre en place (plug-in à installer sur Hadoop) et qu’elle est moins lourde au niveau des traitements (transfert intégrer à la solution alors qu’on utilise des batchs pour la 2e solution). Ce plug-in, MongoDB Connector for Hadoop, permettra de transférer les données de MongoDB sur des tables Hive.

* Pour Oracle, une seule solution s’est présentée : celle d’utiliser Apache Sqoop, un outil permettant de transférer des données entre Hadoop et des entrepôts de données externes tels que Teradata, Oracle, MySQL,etc…

Toutes ces données référentielles seraient mises à jour tous les jours sur les bases origine Oracle et MongoDB, il faudra donc les transférer tous les jours sur des tables Hive. Pas besoin non plus de traiter les données, nous avons simplement besoin de transférer les données sur des tables Hive.

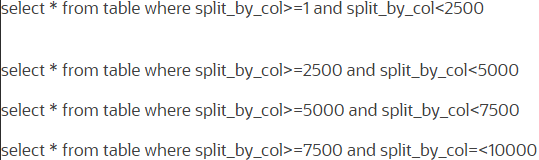
**3-3-2. Implémentation**

*3-3-2.1. Apache Sqoop*

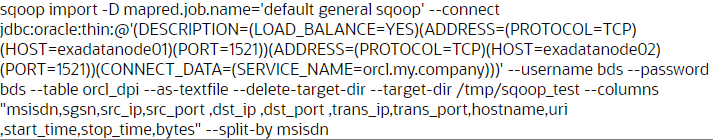
Pour Apache Sqoop, qui va permettre de transférer les données de Oracle à HDFS, le fonctionnement est le suivant : Un job MapReduce va créer une ou plusieurs sessions sur Oracle (qui correspond à un nombre de mappers). Chaque session créée va générer du code SQL et interroger une partie spécifique de la table souhaitée.

Si on exécute un job MapReduce avec un seul mappeur, alors on obtient une seule et unique instruction SQL dans la base de données et Apache Sqoop ne répartit pas le travail sur toute la table. Mais si on veut accélérer ce processus, on peut utiliser plusieurs mappeurs (sessions dans la base de données). Par défaut, Sqoop utilise 4 mappers lorsqu’il requête sur des bases de données externe. Il est possible d’augmenter cette valeur, dans la limite du nombre maximum de cœurs du CPU autorisés à être utilisés par MapReduce sur le cluster.

Une fois le job lancé, Sqoop génère une requête liée à la demande. Par exemple, si on lance une requête sur une table avec 4 mappers (la valeur par défaut), les requêtes suivantes seront créées :



Voilà un exemple d’une commande Sqoop telle qu’elle est utilisée et entrée par l’utilisateur, pour une table nommée “orcl\_dpi” présente sur Oracle et dont la clé primaire serait “msisdn” :



La première ligne permet de créer un job MapReduce et de lui donner un nom par rapport à ce que Sqoop va effectuer. Les lignes 2 à 4 servent à la connexion à la base de données Oracle. Les autres lignes sont là pour apporter les précisions de l’extraction, comme par exemple la destination, le type de format, et les colonnes à extraire de la table.

Dans cet exemple, le nombre de mappers à utiliser n’est pas précisé, donc c’est la valeur par défaut 4 qui sera prise en compte.

Il existe aussi d’autres options que les mappers afin d’optimiser le temps d’exécution d’un job, comme par exemple utiliser la compression de données, qui permet de réduire les temps d’exécution en moyenne de 2 fois.

Il est donc clair que pour notre cas, avec le nombre très important de ligne dans chacunes de nos tables, l’objectif principal sur les commandes Sqoop est de mettre les options d’extraction optimale afin de réduire le temps d’extraction des données.

Nous avons donc tenté de mettre en place des scripts permettant de ramener ces données correctement (script de commande linux afin de trouver le nombre de coeurs optimal à utiliser disponible sur le repo git Deploiement/ReferentialData/FindCore.cmd et script de commande pour Sqoop permettant de ramener les tables sur Hive disponible sur le repo Git Deploiement/ReferentialData/Reseau\_Data.txt)

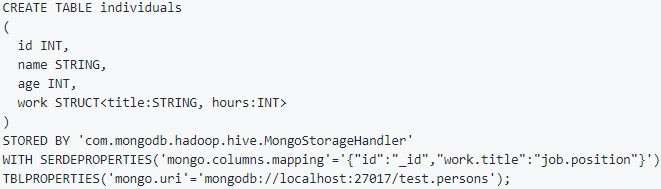
*3-3-2.2. MongoDB Hadoop Connector*

Le plug-in Hadoop qui va nous permettre de transférer les données depuis MongoDB sur des tables Hive fonctionne comme suit :Il faut dans un premier temps télécharger le connecteur général, MongoDB Hadoop Connector, puis des drivers Java afin de pouvoir accéder correctement à MongoDB depuis Hadoop. Enfin, afin d’installer tous ces plug-ins et et driver, on rentre une commande Hive afin pour les inclure dans le fonctionnement de Hive, et ainsi pouvoir transférer les tables depuis MongoDB sur des tables Hive.

Ensuite, il suffit tout simplement de créer des scripts Hive afin de créer, d’insérer des données à partir des données d’une table MongoDB.

La création de tables Hive basées sur MongoDB est gérée par MongoStorageHandler. Il gère également la récupération des données de ces tables (select) et le stockage des données dans ces tables (insert).

Sur l’exemple suivant, on crée une table Hive “individuals” avec 4 colonnes. On précise les liens entre la table d’origine et la table créée : la colonne “id” sur Hive correspond à la colonne “\_id” sur MongoDB, et le champ “work.title” au champ “job.description” de la table MongoDB. Enfin, le nom de la table MongoDB et son emplacement sont précisés.



Nous avons tenté de mettre en place des scripts permettant de ramener ces donénes sur Hive (script de création disponible sur Deploiement/ReferentialData/Deploiement\_Data.hql).

*3-3-2.3. Traitement des données*

Après avoir ramené toutes ces tables, on crée une table globale qui contient les principaux champs qui seront utilisés pour effectuer la consolidation des données par la suite. Les jointures entre les tables s’effectuent à l’intérieur de la requête de création de la table consolidée (script de création disponible sur le repo git /Deploiement/ReferentialData/consolidatedReferential.hql).

## Consolidation des données

### Traitement des données

Le but du batch de consolidation est de permettre aux développeurs utilisant Tableau Server d’avoir le minimum de complexité dans les requêtes à exécuter afin d’extraire les indicateurs du cahier des charges.

Le modèle de donnée que l’on souhaite obtenir est le suivant :

|  |  |  |
| --- | --- | --- |
| Nom colonne | Type | Description |
| pdk | STRING | identifiant du point de concentration |
| date\_indicator | STRING | date de l indicateur, Clé de partition format yyyy-MM-dd |
| nb\_event | INT | Nombre d evenement |
| Commune | STRING | Nom de la commune |
| Departement | STRING | Nom du département |
| Region | STRING | Nom de la région |

Avec une table comme celle-ci, les indicateurs sont très facilement récupérable à moindre coût.

* Nb événements par jour, par type, par équipement

Cet indicateur n’est pas pertinent. En effet, avec 40 millions de compteurs pour seulement quelques millions d’évènements par jour. On voit que la moyenne du nombre d’évènement/jour/compteur est inférieur à 1.

* Nb événements par jour, par type

Select type\_evt, date\_indicator ,nb\_event from consolidatedIndicatorData where date\_indicator = [date] [and(?)] type\_evt = [type]

* Nb événements par jour, par type, par grappe (pdk)

Select type\_evt,date\_indicator,nb\_event from consolidatedIndicatorData where date\_indicator = [date] AND type\_evt = [type] AND pdk = [pdk]

* Nb événements par jour, par type, par commune

Select type\_evt,first(date\_indicator),Commune,sum(nb\_event) from consolidatedIndicatorData where date\_indicator = [date] AND type\_evt = [type] GROUP BY Commune, type\_evt

* Nb événements par jour, par type, par département

Select type\_evt,first(date\_indicator),Departement,sum(nb\_event) from consolidatedIndicatorData where date\_indicator = [date] AND type\_evt = [type] GROUP BY Departement, type\_evt

* Nb événements par jour, par type, par région

Select type\_evt,first(date\_indicator),Region,sum(nb\_event) from consolidatedIndicatorData where date\_indicator = [date] AND type\_evt = [type] GROUP BY Region, type\_evt

Pour une granularité différente :

* Nb événements par semaine, par type

Select type\_evt,sum(nb\_event) from consolidatedIndicatorData where date\_indicator = [date] [and(?)] type\_evt = [type] GROUP BY type\_evt

* Nb événements par semaine, par type, par grappe (pdk)

Select type\_evt, pdk ,sum(nb\_event) from consolidatedIndicatorData where date\_indicator BETWEEN [date1] AND [date2] AND type\_evt = [type] AND pdk = [pdk] GROUP BY pdk,type\_evt

* Nb événements par semaine, par type, par commune

Select type\_evt, Commune,sum(nb\_event) from consolidatedIndicatorData where date\_indicator BETWEEN [date1] AND [date2] AND type\_evt = [type] GROUP BY Commune, type\_evt

* Nb événements par jour, par type, par département

Select type\_evt, Departement,sum(nb\_event) from consolidatedIndicatorData where date\_indicator BETWEEN [date1] AND [date2] AND type\_evt = [type] GROUP BY Departement, type\_evt

* Nb événements par jour, par type, par région

Select type\_evt,first(date\_indicator),Region,sum(nb\_event) from consolidatedIndicatorData where date\_indicator BETWEEN [date1] AND [date2] AND type\_evt = [type] GROUP BY Region, type\_evt

La granularité mensuelle suit la même logique. On peut utiliser l’Opérateur Like en SQL pour récupérer un mois entier.

Toutes les requêtes proposées ont donc une condition sur la clé de partition. Ce qui va améliorer les performances des requêtes.

Il faut donc un batch de consolidation qui va nous permettre de remplir la table consolidatedIndicatorData

### Prototype

Les données de base que nous avons générées sont au format csv. Un premier fichier représente la table eventData et contient 10000 évènements de type EVT\_C, 1000 évènements de type EVT\_K et 1000 évènements de type EVT\_N réparti aléatoirement sur la date de création du fichier – 70 jours pour avoir des données réparties sur plusieurs mois. De plus ils sont attribués aléatoirement aux 400 compteurs du fichier de référence.

Le second fichier représente la table consolidatedReferential. On a 400 compteurs répartis équitablement sur 4 communes dans 3 départements et dans deux régions différentes.

On va utiliser Spark SQL qui s’intègre très bien avec Hive pour le prototype.

Le principe de ce batch est de faire la jointure entre les tables eventData et consolidatedReferential. On fait la jointure sur les id\_equipement/id\_compteur. Elle est réalisée à chaque fois au niveau du jour. Le code permet de faire ça pour les 30 jours précédent le jour de départ. A chaque fois on vient rajouter nos résultats au dataset. Le but sera ensuite de merger ces données dans la table ConsolidatedIndicatorData pour venir ajouter les données du jour et mettre à jour les données des 30 jours précédent. Pour plus de détails, le prototype ainsi que des données de test sont disponibles sur le repo git.