De : NOBLET Gilles

Envoyé : jeudi 6 mai 2021 15:57

À : CATSBG Data Socle Outils et Services <data.socle.outils.et.services@ca-ts.fr>

Cc : RAYER Ugo <Ugo.RAYER@ca-ts.fr>; DAVID Sylvain (EXT) <Sylvain.DAVID-ext@ca-ts.fr>

Objet : RE: BDF - Flux OPEN CREDIT_TRANSFER_TRANSACTION => FRICTIONLESS DATA

Hello Bertrand,

Je rajoute Ugo et Sylvain vu qu'on a bossé sur ces sujets ce matin ensemble.

Extraire des fichiers d'un système source ne devrait idéalement pas se faire à la main surtout si c'est pour les réintégrer de suite vers un autre système, car on s'expose à tous les tracas, mais quand on y est astreint, alors plongeons ensemble dans le tumulte !

Lorsqu'il existe des caractères spéciaux dans les champs, il y a des chances qu'ils soient présents dans la source de données (#1) ou alors que les traitements réalisés sur la source la corrompe in fine (#2). Ainsi, il faudrait savoir si on se trouve dans le cas **(#1)** ou le cas **(#2)**.

Concernant le cas (#1)

L'encodage-client (côté client : SQLA, 3270, MySQL Workbench, dbeaver, etc.) utilisé pour extraire les données de la plateforme doit satisfaire à l'existence d'une relation/correspondance au moins bijective et au mieux injective, de l'encodage côté serveur de cette plateforme vers lui-même, si l'on considère ces jeux de caractères comme des ensembles, et Ô joie ! La traduction anglaise nous donne raison ʃ, => character ***sets.***

Teradata utilise l'UNICODE côté serveur pour identifier les caractères qu'il doit tôt ou tard réémettre vers le client (je ne sais pas comment Teradata stocke ces caractères côté serveur dans la mémoire), ainsi, il faudra au moins un encodage client aussi grand que ce que l'UNICODE permet d'identifier.

e.g. #1.1#a (cas corrompant)

Par exemple, côté client, si on utilise l'encodage-client ASCII pour représenter des chaînes de caractères identifiées par de l'UNICODE (qui est un jeu de caractères et non un encodage), alors, on ne pourra pas représenter tout ce que le serveur nous envoie et on corrompra la donnée dans le

fichier de réception côté client, avant même d'avoir pu continuer vers la situation (#2).

e.g. #1.1#a-bis (cas où l'on s'en sort avec de la chance)

Si toutes vos chaînes de caractères originellement encodées en UTF-8 et chargées ainsi, n'utilisent finalement que des caractères ASCII, alors vous pourrez les extraire en utilisant un character set tel que l'ASCII bien qu'elles aient été originellement chargées via l'UTF-8. Cependant, supposerions-nous que toutes les chaînes de caractères présentes dans le Teradata suivent cette hypothèse ? ⅃

e.g. #1.1#b (cas réputé viable)

Par exemple, côté client, si on utilise l'encodage-client UTF-8 pour représenter des chaînes de caractères identifiées par de l'UNICODE (qui est un jeu de caractères et non un encodage), alors, on pourra représenter ce que le serveur nous envoie et on ne corrompra pas la donnée dans le fichier de réception côté client, néanmoins on peut toujours la corrompre dans le cas (#2).

Il y a d'autres éléments propres au fichier sur lesquels veiller :

#1.2 : Le **délimiteur de champ** (en CSV : RFC4180, c'est une virgule (comma) https://tools.ietf.org/html/rfc4180 or bien des fichiers utilisent **à RAISON,** d'autres caractères moins usités dans la langue humaine dans laquelle les données peuvent s'exprimer, à savoir un PIPE : « | », un caractère TABULATION, ou autre, certains extracteurs doublent ou triplent les délimiteurs de colonne, il est cependant complexe de le faire interpréter à des utilitaires GNU\Linux comme awk via son option –FS (Field Separator).

NOTA BENE : Imaginez que vous deviez importer un fichier avec deux champs, le premier est un ID en DECIMAL(18,0) et l'autre est un champ texte ne comprenant que des lignes de commandes Linux avec donc… la présence de plusieurs caractères PIPE : « | », évidemment, le PIPE n'est pas discriminant dans la langue dans laquelle la donnée s'exprime…

Certains extracteurs définissent des *modes d'extraction*, le mode CSV qui par défaut ajoute des caractères *double quote* à tous les champs ou à ceux qui sont exposés à la présence du délimiteur dans la valeur du champ. C'est ce qu'on va voir dans le point suivant : **#1.3**

#1.3 : Le **délimiteur de champ**, les fameux caractères délimitant le début et la fin d'un champ. Ainsi, qu'importe la présence d'un caractère FIELD SEPARATOR dans un champ de valeur, car c'est ces caractères qui informent où commencer et où terminer l'analyse lexicale et syntaxique (=> parsing) du champ via la grammaire ABNF CSV, Page 3 de la RFC4180.

#1.4 : Le **délimiteur d'enregistrement**, j'emploie ici le terme *enregistrement* à l'instar des bases de données et c'est maladroit, mais c'est pour parler du cas où l'on se trouve avec des MULTI-LINE RECORDs, où **Ed MORTON** nous a gratifiés magnifiquement de sa contribution, brillamment valorisée par **mosvy** ici : https://stackoverflow.com/questions/45420535/whats-the-most-robust-way-to-efficiently-parse-csv-using-awk/45420607#45420607

C'est un cas qui n'est pas si particulier notamment avec des contenus HTML, leurs balises et les caractères FEEDLINE souvent présents dans ces champs où le simple **wc –l** se fait abuser et échoue à nous donner le véritable nombre de « lignes ».

#1.5 : la **présence d'une en-tête** ou non, le fameux HEADER**.**

#1.6 : le **délimiteur de nombres réels** aussi appelés **à TORT** *flottants.* Est-ce une virgule ou un point ou autre ?

#1.7 : l'utilisation de l'écriture scientifique dans les très grands nombres entiers ou réels présentés avec une mantisse ?

…

Il y en a d'autres, n'hésitez pas à amender ou corriger l'existant.

Il y a d'autres éléments propre à la donnée source sur lesquels veiller : (pour valider que l'extraction s'est bien passée…)

#1.a : Le nombre d'enregistrements de la table en source doit correspondre au nombre de lignes du fichier de réception, +/- 1 ligne en considérant le HEADER dans le fichier de réception.

#1.b : le nombre de colonnes dans la table en source doit correspondre au nombre de champs du fichier de réception

#1.c : les types de données utilisés par la structure en source doivent être fournies en métadonnées ainsi que leur format.

#1.d : les colonnes compressées et les algorithmes utilisés par la compression (en Teradata, MVC, BLC, ALC) doivent être aussi fournies :

MVC : Multi-Value Compression, le fameux COMPRESS que vous voyez dans les DDLs

BLC : Block-Level Compression.

ALC : Algorithmic-Level Compression.

#1.e : les colonnes sur lesquelles les contraintes d'intégrité référentielle s'appliquent : PK et FK, MULTISET ou SET table en Teradata.

En effet, une PK définie sur une ou plusieurs colonnes INTERDIT les doublons parfaits et donc apporte une information fortement appréciée dans l'analyse du fichier… Une FK peut aisément être vérifiée avec l'utilitaire **join** en GNU\Linux.

#1.f : les colonnes sur lesquelles les mécanismes d'accès privilégiés à la donnée sont positionnés : PI et son type (No PI, NUPI ou UPI), Index et leur**s** type**s** (USI, SI), Partitions et leurs définitions (Type, nombre de niveau, grain).

#1.g en lien avec le #1.b: toutes les longueurs maximales des champs doivent être fournies afin de renseigner le bon dimensionnement des types de données à choisir pour la plateforme cible et en vérifiant que les types en source ne sont pas violés lors de l'extraction, e.g. un VARCHAR(64) CASESPECIFIC LATIN ou UNICODE fait combien en longueur maximale ? ;-)

…

Il y en a d'autres, n'hésitez pas à amender ou corriger l'existant.

Concernant le cas (#2)

Enfin le cas (#2) ! On a désormais vérifié que notre fichier est bien extrait et on le répute viable au regard de la source.

Et c'est là que les joyeusetés commencent. On doit désormais intégrer la donnée dans le système cible.

Chaque développeur va tôt ou tard s'exprimer de la manière la plus rigoureuse et la plus aboutie pour gérer un « cas particulier » qu'il a trouvé avec ses pratiques, ses outils, son expérience, etc. et des cas particuliers, il y en a… beaucoup, beaucoup trop d'ailleurs… et chacun exprime sa meilleure solution qui ne considère malheureusement jamais toute la complexité des autres situations gérées par les collègues. **C'est la situation dans laquelle nous sommes tombés ce matin avec Sylvain DAVID.**

Ces fameux cas doivent être comme d'habitude reproductibles, documentés, versionnés et enseignés à tous les développeurs et cela ne peut se faire autrement que via un cadre, qui dit cadre, dit cadriciel, dit framework et pour se faire, comment faire ?

Frictionless Data

Parlons donc d'un cadriciel dans lequel s'assurer que tous ces points sont vérifiés, **Frictionless Data :** https://frictionlessdata.io/

C'est une solution que j'utilisais depuis quelques temps, à vous de me donner votre avis lorsque vous la prendrez en main si vous avez le temps.

C'est une solution qui s'attarde à standardiser la transformation de données et c'est plus que plaisant car on sait les cas traités et ceux non traités qui nous restent à développer nous-mêmes. Tout ce qu'on a dit précédemment fait évidemment partie de notre base de connaissances tôt ou tard consignées dans une documentation 3U pour rapidement monter en compétences chaque développeur et au moins lui offrir une aptitude à manipuler la donnée via les utilitaires Linux, les Perl/Python distributions présentes déjà sur les serveurs PROD/HORS-PROD mais ce n'est jamais cette connaissance diverse, multi-niveau et mono-implémentée qui nous sert à fiablement réputer nos données saines ou corrompues, c'est le framework qui assure ce travail.

De : CATSBG Data Socle Outils et Services <data.socle.outils.et.services@ca-ts.fr>

Envoyé : jeudi 6 mai 2021 12:10

À : NOBLET Gilles <Gilles.NOBLET@ca-ts.fr>

Objet : TR: BDF - Flux OPEN CREDIT_TRANSFER_TRANSACTION

Gilles,

Pour ce que je vois cette fonction permet de virer des caractères (genre saut de ligne) dans les zones de texte. Sais-tu quelle correction devrait y être apporter pour qu'elle marche ?

Bertrand

De : DAVID Sylvain (EXT) <Sylvain.DAVID-ext@ca-ts.fr>

Envoyé : jeudi 6 mai 2021 11:43

À : CATSBG Data Socle Outils et Services <data.socle.outils.et.services@ca-ts.fr>

Cc : RAYER Ugo <Ugo.RAYER@ca-ts.fr>; CATSBG Data metier Bancaire <data.metier.bancaire@ca-ts.fr>

Objet : RE: BDF - Flux OPEN CREDIT_TRANSFER_TRANSACTION

Après analyse avec Gilles nous avons trouvé la raison de l'ajout des « : ».

Cela provient de la fonction drop_CR_within_Dbl_Quote présente dans le fichier /opt/MUP10/Logiciel/{AppliCDP}/CDP/OPN/shl .

Dans cette fonction, on fait la commande suivante :

awk -v FS="${SEP}" -v RS='"[^"]*"' -v ORS= '{ gsub(FS, ":", RT); gsub(/\n/, " ", RT); gsub(/\r/, " ", RT); print $0 RT}' ${FILE_NAME_TMP} > ${DIR_PATH}/${FILE_NAME}.tmp

Cette commande remplace le dernier champ du fichier par « : ».

En l'état, un fichier pour un CDP OPEN ne peut donc jamais terminer par une colonne nullable et spécialement une dernière ligne avec le champ vide.

Cette fonction est appelée dans tous les flux OPEN.

Après avoir discuté avec Gilles, il faudrait faire porter l'ensemble des transformations au TPT plutôt qu'à des fonctions shell annexes.

Pourriez-vous voir, dans le cadre des travaux sur le nouveau générateur, pour corriger ce problème svp ?

Pour pallier à notre souci, je vais donc devoir modifier le shell généré pour mettre en commentaire l'appel de cette fonction.

Sylvain DAVID

De : DAVID Sylvain (EXT)

Envoyé : jeudi 6 mai 2021 09:44

À :

Cc : RAYER Ugo <Ugo.RAYER@ca-ts.fr>; CATSBG Data metier Bancaire <data.metier.bancaire@ca-ts.fr>

Objet : BDF - Flux OPEN CREDIT_TRANSFER_TRANSACTION

Dans le cadre du projet Carto BDF, nous avons un flux OPEN CREDIT_TRANSFER_TRANSACTION à mettre en place.

Nous avons effectué le développement et testé le flux avec un fichier fournit par le bloc amont.

Cependant, nous rencontrons un problème : la dernière colonne du fichier peut être null et le traitement TPT renseigné dans la dernière ligne du fichier pour cette colonne la valeur « : ».

Cette colonne représente un TIMESTAMP, cette valeur provoque donc le plantage de notre flux pour « Invalid timestamp ».

Il n'y a pas de raison d'avoir ces « : » puisque dans le fichier, il n'y aucune valeur (vérification qu'il n'y a pas de caractères spéciaux cachés).

De plus, le problème se produit que nous ne mettions qu'une seule ligne dans le fichier ou bien l'ensemble du JDD.

Est-ce que vous auriez une explication svp ? Un fichier OPEN peut-il terminer avec une colonne nullable ?

Les données sont dispos dans la I1 :
DZUDA0CPDT_WRK_MU_0.CREDIT_TRANSFER_TRANSACTION_I1

Cordialement,

Sylvain DAVID

https://stackoverflow.com/questions/45420535/whats-the-most-robust-way-to-efficiently-parse-csv-using-awk/45420607#45420607

What's the most robust way to efficiently parse CSV using awk?

Given a CSV as might be generated by Excel or other tools with embedded newlines, embedded double quotes and empty fields like:

```
$ cat file.csv
"rec1, fld1",,"rec1"",""fld3.1
"",
fld3.2","rec1
fld4"
"rec2, fld1.1

fld1.2","rec2 fld2.1""fld2.2""fld2.3"","",rec2 fld4
```

What's the most robust way efficiently using awk to identify the separate records and fields:

```
Record 1:
    $1=<rec1, fld1>
    $2=<>
    $3=<rec1","fld3.1
",
fld3.2>
    $4=<rec1
fld4>
----
Record 2:
    $1=<rec2, fld1.1
```

```
fld1.2>
    $2=<rec2 fld2.1"fld2.2"fld2.3>
    $3=<>
    $4=<rec2 fld4>
----
```

so it can be used as those records and fields internally by the rest of the awk script.

A valid CSV would be one that conforms to RFC 4180 or can be generated by MS-Excel.

The solution must tolerate the end of record just being LF (`\n`) as is typical for UNIX files rather than CRLF (`\r\n`) as that standard requires and Excel or other Windows tools would generate. It will also tolerate unquoted fields mixed with quoted fields. It will specifically not need to tolerate escaping "s with a preceding backslash (i.e. `\"` instead of `""`) as some other CSV formats allow - if you have that then adding a `gsub(/\\"/,"\"\"")` up front would handle it and trying to handle both escaping mechanisms automatically in one script would make the script unnecessarily fragile and complicated.

CommunityBot

Ed Morton

If your CSV cannot contain newlines or escaped double quotes then all you need is (with GNU awk for FPAT):

```
$ echo 'foo,"field,with,commas",bar' |
    awk -v FPAT='[^,]*|"[^"]+"' '{for (i=1; i<=NF;i++) print i, "<" $i ">"}'
1 <foo>
2 <"field,with,commas">
3 <bar>
```

If all you actually want to do is convert your CSV to individual lines by, say, replacing newlines with blanks and commas with semi-colons inside quoted fields then all you need is this, again using GNU awk for multi-char RS and RT:

```
$ awk -v RS='"[^"]+"' -v ORS= '{gsub(/\n/," ",RT); gsub(/,/,";",RT); print $0
RT}' file
"rec1; fld1",,"rec1"";""fld3.1 ""; fld3.2","rec1 fld4"
"rec2; fld1.1  fld1.2","rec2 fld2.1""fld2.2""fld2.3","",rec2 fld4
```

Otherwise, though, the general, robust, portable solution to identify the fields that will work with any modern awk is:

```
$ cat decsv.awk
function buildRec(      i,orig,fpat,done) {
    $0 = PrevSeg $0
    if ( gsub(/"/,"&") % 2 ) {
        PrevSeg = $0 RS
        done = 0
    }
    else {
        PrevSeg = ""
```

```
        gsub(/@/,"@A"); gsub(/""/,"@B")             # <"x@foo""bar"> ->
<"x@Afoo@Bbar">
        orig = $0; $0 = ""                          # Save $0 and empty it
        fpat = "([^" FS "]*)|(\"[^\"]+\")"          # Mimic GNU awk FPAT meaning
        while ( (orig!="") && match(orig,fpat) ) { # Find the next string
matching fpat
            $(++i) = substr(orig,RSTART,RLENGTH)   # Create a field in new $0
            gsub(/@B/,"\"",$i); gsub(/@A/,"@",$i)  # <"x@Afoo@Bbar"> ->
<"x@foo"bar">
            gsub(/^"|"$/,"",$i)                    # <"x@foo"bar">   ->
<x@foo"bar>
            orig = substr(orig,RSTART+RLENGTH+1)   # Move past fpat+sep in orig
$0
        }
        done = 1
    }
    return done
}

BEGIN { FS=OFS="," }
!buildRec() { next }
{
    printf "Record %d:\n", ++recNr
    for (i=1;i<=NF;i++) {
        # To replace newlines with blanks add gsub(/\n/," ",$i) here
        printf "    $%d=<%s>\n", i, $i
    }
    print "----"
}


.

$ awk -f decsv.awk file.csv
Record 1:
    $1=<rec1, fld1>
    $2=<>
    $3=<rec1","fld3.1
",
fld3.2>
    $4=<rec1
fld4>
----
Record 2:
    $1=<rec2, fld1.1

fld1.2>
    $2=<rec2 fld2.1"fld2.2"fld2.3>
    $3=<>
    $4=<rec2 fld4>
----
```

The above assumes UNIX line endings of `\n`. With Windows `\r\n` line endings it's much simpler as the "newlines" within each field will actually just be line feeds (i.e. `\n`s) and so you can set `RS="\r\n"` (using GNU awk for multi-char RS) and then the `\n`s within fields will not be treated as line endings.

It works by simply counting how many `"`s are present so far in the current record whenever it encounters the `RS` - if it's an odd number then the `RS` (presumably `\n` but doesn't have to be) is mid-field and so we keep building the current record but if it's even then it's the end of the current record and so we can continue with the rest of the script processing the now complete record.

The `gsub(/@/,"@A"); gsub(/""/,"@B")` converts every pair of double quotes axcross the whole record (bear in mind these `""` pairs can only apply within quoted fields) to a string `@B` that does not contain a double quote so that when we split the record into fields the match() doesn't get tripped up by quotes appearing inside fields. The `gsub(/@B/,"\"",$i); gsub(/@A/,"@",$i)` restores the quotes inside each field individually and also converts the `""`s to the `"`s they really represent.

Also see How do I use awk under cygwin to print fields from an excel spreadsheet? for how to generate CSVs from Excel spreadsheets.

edited Dec 29 2021 at 21:12
answered Jul 31 2017 at 16:06

Ed Morton

An improvement upon @EdMorton's `FPAT` solution, which should be able to handle double-quotes(") escaped by doubling ("" -- as allowed by the CSV standard).

```
gawk -v FPAT='[^,]*|("[^"]*")+' ...
```

This STILL

1. isn't able to handle newlines inside quoted fields, which are perfectly legit in standard CSV files.
2. assumes **GNU awk** (`gawk`), a standard awk won't do.

Example:

```
$ echo 'a,,"","y""ck","""x,y,z"," ",12' |
gawk -v OFS='|' -v FPAT='[^,]*|("[^"]*")+' '{$1=$1}1'
a||""|"y""ck"|"""x,y,z"|" "|12

$ echo 'a,,"","y""ck","""x,y,z"," ",12' |
gawk -v FPAT='[^,]*|("[^"]*")+' '{
  for(i=1; i<=NF;i++){
    if($i~/"/){ $i = substr($i, 2, length($i)-2); gsub(/""/,"\"", $i) }
    print "<"$i">"
  }
}'
<a>
<>
<>
<y"ck>
<"x,y,z>
< >
<12>
```

mosvy

> That's a nice FPAT, I can't imagine any invalid case it'd allow - you should suggest the gawk folks update their FPAT documentation to use it instead of `FPAT = "([^,]*)|(\"[^\"]+\")` as documented at the bottom of that section and I used.
> – Ed Morton

Mar 13 2020 at 20:02

This is exactly what csvquote is for - it makes things simple for awk and other command line data processing tools.

Some things are difficult to express in awk. Instead of running a single awk command and trying to get awk to handle the quoted fields with embedded commas and newlines, the data gets prepared for awk by csvquote, so that awk can always interpret the commas and newlines it finds as field separators and record separators. This makes the awk part of the pipeline simpler. Once awk is done with the data, it goes back through `csvquote -u` to restore the embedded commas and newlines inside quoted fields.

```
csvquote file.csv | awk -f my_awk_script | csvquote -u
```

D Bro

- Please edit your answer to include the output of `csvquote file.csv` so we can see what exactly it'd do to the CSV from my question. Also add the output of `csvquote file.csv | csvquote -u` so we can see if it reproduces the input unchanged.
  – Ed Morton
  Jan 31 at 15:22