Unicage **Open version**

For Developers

Quick guide

Contents

1. UNICAGE OPEN VERSION

What is Unicage Open Version?

Unicage Open Version is a set of 85 Unicage commands that are freely distributed and that can be easily installed on any machine with a Linux environment. Contrarily to the ones distributed through Unicage Enterprise Version, these commands are written in Python, therefore, they are not able to achieve the same performance levels as the ones from the Unicage Enterprise Version, and they might present certain limitations in terms of execution.

Requirements

To install Unicage Open Version, you just need a Linux Operating System and Python 2.4 or above.

Install

To install the Unicage Open Version commands, you just need to follow the following steps:

1) Download and extract the repository from Github (https://unicage.eu/unicage-open-version/).
2) Open your terminal inside the extracted folder and type: ***sudo make install*** This will install the Unicage commands in the default path /usr/local/bin.

After the previous steps are executed, you can now use the Unicage commands by simply using the command line or through Shell scripts.

Test

To check if the installation was successful, you can execute the following command: ***sudo make test*** This command will execute a series of scripts that will check if the Unicage commands were installed correctly in your system. Please note that these tests might take a while to finish.

Uninstall

If you want to remove the Unicage Open Version from your computer, just go to the directory containing the data you downloaded from the GitHub repository, open your terminal and type: ***sudo make uninstall*** The commands should now be removed from your system.

2. UNICAGE AND IT'S METHODOLOGY

Unicage is a set of highly efficient commands that allow the user to build robust, yet flexible systems in a modular way through data processing pipelines. Unicage follows the Unix philosophy, a set of concepts and guidelines that focus on designing small but highly efficient programs and operating systems.

This philosophy can be summarized in 8 core topics:

- Small is beautiful.
- One program (command) should only do one thing.
- Prototyping should be as fast as possible.
- Portability takes precedence over efficiency.
- Data is stored as plain text.
- Commands are used as "levers" (can be combined & reused).
- Applications are written in shell script.
- All programs are designed as filters (pipes).

2.1. Why is the processing speed so fast?

Each command of the Unicage Enterprise version is written in C language, and the input/output buffer, memory manipulation and calculation algorithm have been designed to allow high-speed processing.

The Shell uses kernel functions directly. By removing middleware there is no processing. Unicage Shell Script programming methodology avoids slow variable type programming and functional programming, and it follows the data flow programming that takes advantage of the processing speed of each command.

In Unicage Shell programming we organize the data in advance for increased performance. Unicage developed high-speed commands for complex sorting.

References:

https://dl.acm.org/doi/10.1145/3136014.3136031
https://benchmarksgame-team.pages.debian.net/benchmarksgame/
https://github.com/niklas-heer/speed-comparison
http://www.hildstrom.com/projects/langcomp/index.html
http://attractivechaos.github.io/plb/

2.2. Comparison with mainstream technologies

Performance benchmark vs mainstream data storage and data processing frameworks (Spark, SparkSQL, Kudu/ HDFS, Hadoop)
Research studies in prestigious higher education institutions both in the United States (MIT), Japan (Kanazawa University) and Europe (IST Lisbon) show results ranging from 3 to 50 times faster.

Gains in development productivity
Unicage provides a significant reduction of lines of code (depending on the language to be converted, for example Cobol application re-writing is a 20:1 ratio). Unicage is also easy to read and understand and provides easily measurable auditing to the usage of the system.

2.3. How is the data organized at the storage level (column, row)?

Data is stored in a UNIX file system as a regular flat text file format. The Unicage methodology consists of the way data is organized and then processed by our proprietary commands. The uniqueness of the solution lies on its ability to utilize flat text files instead of requiring middleware or relational database engines that decrease the speed of execution.

2.4. How is the data distributed across data units?

Parallel processing is done on a master-slave model where flat text files are divided according to the logic of the script across all the available nodes. Those nodes execute the script and finally they are merged and consolidated into a new text file with the result of the execution.

2.5. Data file management (levels)

Unicage organizes data files into business units along five levels ('OSAHO' – Unicage's etiquette of data file management)

- Level 1 (event data)
- Level 2 (confirmed data)
- Level 3 (organized data)
- Level 4 (application reference data
- Level 5 (application output data)

Unicage does not require the existence of any special file management software

2.6. Interoperability, locking, compression and memory management

Interoperability
Unicage is based on UNIX fundamentals. Interoperability through frameworks such as Tivoli (IBM) or JP1 (Hitachi).

Locking
Unicage does not require locking unless a concurrent situation is present. For those purposes, the command "*ulock*".

Compression
Unicage is compatible with multiple compression tools. For example .gz .Z is used for data compression.

Memory Management
Unicage processes data based on streaming, which decreases the memory usage comparatively to technologies such as java or python. Managing memory is accomplished by Unix commands.

2.7. Security (authentication, authorization and encryption)

Unicage utilizes segregation mechanisms of the underlying UNIX Operating System: filesystem permissions, memory stack protection and role-based access controls.

Encryption can be achieved on several levels - either native filesystem encryption mechanisms (F2FS in Linux or ZFS in BSD/UNIX) or 3rd party mechanisms offered by several vendors. Self-encrypting disk is also a possibility as Unicage just uses the Operating System POSIX infrastructure to access the data storage.

Security can be increased through File checksumming tools (such as native capacity or products like Tripwire) and rule-based firewall.

A typical Linux/UNIX node running Unicage will only need SSH as an open port - this can be ensured by service minimization (disabling and/or uninstalling unnecessary services). This is implemented by a host-based firewall, which will permit access only from known hosts. SSH access is restricted to a number of known users (no Administration/Super User access is conceded).

Essential configuration files are then protected by checksumming to ensure no alteration of content.

File systems can be encrypted to prevent data loss and processes will run only with needed privileges inside protected memory.

2.8. UNIX parallel processing

UNIX is a multi-user, multi-tasking OS. You can run multiple jobs for multiple users at the same time.

- Parallel processing commands used:
- Specify “& (background)“ when the job starts, to parallelize the job
- “bg” or ”fg” commands switch between parallelizing and sequencing along the processing
- “nice“ command changes the priority of parallel processing
- ”stop“ or ”kill” commands interrupt or stop the job
- “jobs” or “ps” or “tree” commands allow monitoring the parallel processing

Above mentioned job control commands, allow for writing a shell script to perform parallel processing in any number of processes.

2.9. Partitioning, indexing and concurrency

Partitioning
There are no special requirements on partitioning. Nodes can be independent servers or virtualized (i.e Docker). The only recommendation we provide is to leave 10% disk space available for our data operations inside the disk.

Indexing
There are no special requirements for indexing as everything is executed on the UNIX file system as a text file.

Concurrency
Unicage is based on UNIX fundamentals where multithreading, concurrent and exclusive processes are allowed. Our suite of commands contains blocking commands as well as atomic writing.

2.10. Scalability and workload management support

Scalability
Unicage scales quasi-linearly with extra hardware. The cluster version commands provide an automatic map and reduce process.

Workload Management support
There are multiple frameworks that control UNIX processes. As an example, for general workload management commands such as “ulimit” or “nice” can be used. Enforcing detail management usually is handled by sub-OS functions such as “cgroup” or “jail”.

3. COMMANDS AND EXAMPLES

In this section, we give you examples and descriptions of some common and most used Unicage commands that come with the Unicage Open Version. If you want to know more about these commands or other commands included in the Unicage Open Version, just type ***man2 command_name*** in your terminal and the respective manual page will be displayed, with an explanation regarding the command along with some usage examples.

msort

Sorts a file according to a given key, which will correspond to a field or set of fields.

There is no limit on key length or number of key fields, and it does not matter if the file contains multi-byte characters such as Japanese.

You can use **r** after a field position sorts that field in descending order.

If you specify **n** after the field position, that field's values will be sorted as numbers. If you specify **N** after the field, the values will be sorted in descending order as numbers. You can also use **nr** to compare numerically and sort in descending order.

sample

```
002 023 A 20210103
001 025 C 20210103
003 012 B 20210105
002 022 C 20210102
```

msort key=1/2 sample	msort key=3@1 sample	msort key=2n sample
001 025 C 20210103	002 023 A 20210103	003 012 B 20210105
002 022 C 20210102	003 012 B 20210105	002 022 C 20210102
002 023 A 20210103	001 025 C 20210103	002 023 A 20210103
003 012 B 20210105	002 022 C 20210102	001 025 C 20210103

self

Allows you to reorder fields, select substrings from fields and/or filter content in a file.
By using ***-d*** option, you can execute the command on a string.

self field1 field2 … fieldN <file>

self -d field1 field2 … fieldN <string>

sample

002 023 A 20210103

001 025 C 20210103

003 012 B 20210105

002 022 C 20210102

self 3 1 2 sample	self 3 4 sample	self 3 1 4.1.4 4.5.2 4.7
A 002 023	A 20210103	A 002 2021 01 03
C 001 025	C 20210103	C 001 2021 01 03
B 003 012	B 20210105	B 003 2021 01 05
C 002 022	C 20210102	C 002 2021 01 02

delf

Deletes fields from a file.

delf f1 f2 … fN <file>

sample

002 023 A 20210103

001 025 C 20210103

003 012 B 20210105

002 022 C 20210102

delf 2 sample	delf 2/3 sample
002 A 20210103	002 20210103
001 C 20210103	001 20210103
003 B 20210105	003 20210105
002 C 20210102	002 20210102

selr

Selects the rows of the file that match a given string in a specific field.

selr <field> <string> <file>

sample

002 023 A 20210103

001 025 C 20210103

003 012 B 20210105

002 022 C 20210102

selr 3 C sample	selr 1 002 sample
001 025 C 20210103	002 023 A 20210103
002 022 C 20210102	002 022 C 20210102

delr

Deletes the rows of the file that match a given string in a specific field.

delr <field> <string> <file>

sample

002 023 A 20210103

001 025 C 20210103

003 012 B 20210105

002 022 C 20210102

delr 3 C sample	delr 1 002 sample
002 023 A 20210103 003 012 B 20210105	001 025 C 20210103 003 012 B 20210105

sm2

Sums the elements of specified fields based on the key-fields chosen by the user. The first two arguments correspond to the range of the fields the key occupies, whilst the last two arguments correspond to the range of fields that will be summed.
By using the ***+count*** option, you can obtain the total number of records that were summed.

sm2 <key1> <key2> <field1> <field2> <file>

sample_sales

001 Agata 20110405 32 1600

001 Agata 20110406 24 1200

001 Agata 20110407 49 2450

002 Tony 20110405 102 5100

002 Tony 20110406 98 4900

002 Tony 20110407 121 6050

sm2 1 2 4 5 sample_sales	sm2 ***+count*** 1 2 4 5 sample_sales
001 Agata 105 5250 002 Tony 321 16050	001 Agata 3 105 5250 002 Tony 3 321 16050

sm4

Sums the elements of specified fields based on the key-fields chosen by the user, displaying the subtotals. The first two arguments correspond to the range of the fields the key occupies, the third and fourth arguments correspond to the range of fields to be ignored, and the last two arguments correspond to the range of fields that will be summed.

sm4 <key1> <key2> <field_to_ignore1> <field_to_ignore_2> <field_to_sum1> <field_to_sum2> <file>

sample_sales

001 Agata 20110405 32 1600

001 Agata 20110406 24 1200

001 Agata 20110407 49 2450

002 Tony 20110405 102 5100

002 Tony 20110406 98 4900

002 Tony 20110407 121 6050

sm4 1 2 3 3 4 5 sample_sales

001 Agata 20110405 32 1600
001 Agata 20110406 24 1200
001 Agata 20110407 49 2450
001 Agata @@@@@@@@ 105 5250
002 Tony 20110405 102 5100
002 Tony 20110406 98 4900
002 Tony 20110407 121 6050
002 Tony @@@@@@@@ 321 16050

sm5

Sums the elements of specified fields based on ignoring the key, thus obtaining a grand total. The first two arguments correspond to the range of the fields to be ignored, whilst the last two arguments correspond to the range of fields that will be summed

sm5 <field_to_ignore1> <field_to_ignore_2> <field_to_sum1> <field_to_sum2> <file>

sample_sales

```
001 Agata 20110405 32 1600
001 Agata 20110406 24 1200
001 Agata 20110407 49 2450
002 Tony 20110405 102 5100
002 Tony 20110406 98 4900
002 Tony 20110407 121 6050
```

```
sm5 1 3 4 5 sample_sales

001 Agata 20110405 32 1600
001 Agata 20110406 24 1200
001 Agata 20110407 49 2450
002 Tony 20110405 102 5100
002 Tony 20110406 98 4900
002 Tony 20110407 121 6050
@@@ @@@@@ @@@@@@@@ 426 21300
```

ratio

Gives the percentage of a value in the field based on the overall sum of all values of that field.

ratio [key=<K>] val=<V> <file>

sample_shop_sales

```
ShopA 20210104 10 1000
ShopA 20210105 8 800
ShopA 20210106 7 700
ShopB 20210104 5 500
ShopB 20210105 7 700
ShopB 20210106 8 800
```

```
ratio key=1 val=3/4

ShopA 20210104 10 40.0 1000 40.0
ShopA 20210105 8 32.0 800 32.0
ShopA 20210106 7 28.0 700 28.0
ShopB 20210104 5 25.0 500 25.0
ShopB 20210105 7 35.0 700 35.0
ShopB 20210106 8 40.0 800 40.0
```

It is possible to set the number of decimal places by using the option ***-N***, where **N** corresponds to the number of decimal places.

```
ratio -0 key=1 val=3/4

ShopA 20210104 10 40 1000 40
ShopA 20210105 8 32 800 32
ShopA 20210106 7 28 700 28
ShopB 20210104 5 25 500 25
ShopB 20210105 7 35 700 35
ShopB 20210106 8 40 800 40
```

join0

Extracts from the Transaction File only the records that contain items present in the Master File. The items in the Transaction File need to be sorted before applying the join0 command.

join0 [+ng] key=<K> <MASTER> <TRANSACTION>

LIST_MASTER

A 001 Billy

A 002 Dilan

B 002 Dora

B 003 Anne

RESULTS_TRANS

A 001 330

A 002 450

A 003 345

B 001 320

B 002 403

B 003 409

B 004 381

```
join0 key=1/2 LIST_MASTER RESULTS_TRANS

A 001 330
A 002 450
B 002 403
B 003 409
```

By attaching the option ***+ng*** to the join0, one can extract the records that are not present in the Master File to standard output. To separate the records, redirection must be used.

```
join0 +ng key=1/2 LIST_MASTER RESULTS_TRANS > /dev/null

A 003 345
B 001 320
B 004 381
```

You can separate the records to two distinct files by doing the following:

join0 ***+ng*** key=1/2 LIST_MASTER RESULTS_TRANS ***> RESULTS_OK 2> RESULTS_NOK***

In the RESULTS_OK file, the records from the Transaction File that are present in the Master File will be stored and in RESULTS_NOK file, the records from the Transaction File that are not present in the Master file will be stored.

join1

Similar to join0, but it adds to the Transaction File additional information that is present in the Master's File fields.

join1 [+ng] key=<K> <MASTER> <TRANSACTION>

LIST_MASTER

A 001 Billy
A 002 Dilan
B 002 Dora
B 003 Anne

RESULTS_TRANS

A 001 330
A 002 450
A 003 345
B 001 320
B 002 403
B 003 409
B 004 381

```
join1 key=1/2 LIST_MASTER RESULTS_TRANS

A 001 Billy 330
A 002 Dilan 450
B 002 Dora 403
B 003 Anne 409
```

join2

Similar to join1, but it also outputs the fields that are not present in the Master File in a special format.
join2 [+ng] key=<K> <MASTER> <TRANSACTION>

LIST_MASTER

A 001 Billy
A 002 Dilan
B 002 Dora
B 003 Anne

RESULTS_TRANS

A 001 330
A 002 450
A 003 345
B 001 320
B 002 403
B 003 409
B 004 381

```
join2 key=1/2 LIST_MASTER RESULTS_TRANS

A 001 Billy 330
A 002 Dilan 450
A 003 _ 345
B 001 _ 320
B 002 Dora 403
B 003 Anne 409
B 004 _ 381
```

By default, the fields that are not present in the Master File will have a '_' in their place. However, you can change this value by using the ***+*** option and specifying the string that you want in front.

join2 [+string] [+ng] key=<K> <MASTER> <TRANSACTION>

```
join2 +other key=1/2 LIST_MASTER RESULTS_TRANS

A 001 Billy 330
A 002 Dilan 450
A 003 other 345
B 001 other 320
B 002 Dora 403
B 003 Anne 409
B 004 other 381
```

cjoin0, cjoin1, cjoin2

The cjoin0, cjoin1 and cjoin2 commands are equivalent to the join0, join1 and join2 commands, respectively. However, the cjoin commands do not require the Transaction File to be sorted.

Since the cjoin commands load the entire Master File into memory, the usage of the cjoin commands is recommended over the join commands when you have a small Master File and extremely large Transaction Files.

loopj

Links multiple files with the same key. The files need to be sorted by key.
If there are records with non-existent values in some files, by default a “0” will be placed. This character can be changed by using the option ***-d*** and specifying the character that you want in front of it.

loopj [-d<string>] num=<K> <file_1> <file_2> … <file_N>

SAMPLE1

001 Oslo
002 Lisbon
003 Paris
004 Madrid

SAMPLE2

001 50320 324.1
002 2199 4.3
003 3310 134.8
004 2201 6.4

SAMPLE3

001 NOR
002 POR
003 FRN

loopj ***-d@*** num=1 SAMPLE1 SAMPLE2 SAMPLE3

001 Oslo 50320 324.1 NOR
002 Lisbon 2199 4.3 POR
003 Paris 3310 134.8 FRN
004 Madrid 2201 6.4 @

up3

Merges two files that have the same key. Both files need to be sorted before using up3.

up3 key=<K> <file_1> <file_2>

SAMPLE1

001 Oslo
002 Lisbon
003 Paris
004 Madrid

SAMPLE2

001 50320 324.1
002 2199 4.3
003 3310 134.8
004 2201 6.4

```
up3 key=1 SAMPLE1 SAMPLE2

001 Oslo 50320 324.1
002 Lisbon 2199 4.3
003 Paris 3310 134.8
004 Madrid 2201 6.4
```

getfirst, getlast

getfirst fetches the first records with a given key in a file, while getlast fetches the last records with a given key in a file. The file needs to be sorted by the key before using these commands.

It is possible to use the ***+ng*** option to fetch all records except the first or last, depending on the command used.

getfirst [+ng] <K1> <K2> <file>

getlast [+ng] <K1> <K2> <file>

SAMPLE

```
Japan Chiba 100
Japan Chiba 90
Japan Tokyo 200
Portugal Lisbon 120
Portugal Lisbon 100
Portugal Porto 80
```

```
getfirst 1 1 SAMPLE

Japan Chiba 100
Portugal Lisbon 120
```

```
getlast 1 1 SAMPLE

Japan Tokyo 200
Portugal Porto 80
```

ctail

Removes the last N records from a file.

ctail -<N> <file>

STATES

001 Maine

002 Vermont

003 New_York

004 Delaware

005 Georgia

ctail -2 STATES

001 Maine
002 Vermont
003 New_York

count

Counts the number of records with the same key within a file.

count <K1> <K2> <file>

SAMPLE

A

A

A

B

count 1 1 SAMPLE

A 3
B 1

rank

Counts the occurrences and associates them consecutive numbers starting in 1. By using ***ref*** option, it is possible to associate the numbers based on the given key.

rank [ref=<K>] <file>

SAMPLE

```
001 Tony
001 Gina
001 Anne
002 Carl
003 Sara
```

```
rank SAMPLE
1 001 Tony
2 001 Gina
3 001 Anne
4 002 Carl
5 003 Sara
```

```
rank ref=1 SAMPLE
1 001 Tony
2 001 Gina
3 001 Anne
1 002 Carl
1 003 Sara
```

map

Converts records with repeated keys into a table of records where the repeated keys are intertwined in a way that the information becomes more perceptible. In a way, it converts the records from a format of < x, y, value > into a matrix of x per y. If there are fields without values, a 0 will be placed instead. This value can be changed with the ***-m*** option and by specifying the character.

map [-m<C>] [+yarr] num=<N> <file>

SALES

```
20110201 SHOP_A 13
20110201 SHOP_B 34
20110202 SHOP_A 20
20110202 SHOP_B 18
```

```
map num=1 SALES
*           SHOP_A SHOP_B
20110201    13     34
20110202    20     18
```

The ***+yarr*** option adds an additional horizontal axis.

SALES_JAPAN

```
SHOP_A Tokyo 2016 1354 2135 1255
SHOP_A Tokyo 2017 2133 1245 4265
SHOP_B Osaka 2016 984 824 793
SHOP_B Osaka 2017 908 1193 3145
```

```
map num=2 SALES_JAPAN

*          *       *    2016   2017
SHOP_A   Tokyo    A    1354    2133
SHOP_A   Tokyo    B    2135    1245
SHOP_A   Tokyo    C    1255    4265
SHOP_B   Osaka    A    984     908
SHOP_B   Osaka    B    824     1193
SHOP_B   Osaka    C    793     3145


map +yarr num=2 SALES_JAPAN

*          *       2016  2016  2016  2017  2017  2017
*          *         a     b     c     a     b     c
SHOP_A   Tokyo     1354  2135  1255  2133  1245  4265
SHOP_B   Osaka     984   824   793   908   1193  3145
```

calsed

String substitution by using a customized version of sed. It does not support regular expressions. If the string is replaced by a "@" character, then the ***-nx*** option must be used. If you want to replace a given character C by a blank space, then the ***-s*** option must be used.

calsed [-s<C>] [-nx] <ORIGINAL_STRING> <REPLACEMENT_STRING> <file>

SAMPLE

Tokyo

Lisbon

Milan

calsed Milan Rome SAMPLE

Tokyo
Lisbon
Rome

calsed ***-nx*** Milan @ SAMPLE

Tokyo
Lisbon
@

calsed ***-s_*** Lisbon La_Paz SAMPLE

Tokyo
La Paz
Milan

fsed

Replaces a string on a given field of each record. Supports regular expressions through the ***-e*** option.

fsed [-e] ‘s/<ORIGINAL_STRING>/<REPLACEMENT_STRING>/<field_number>’ <file>

SCORES

```
000149 Lyn 18 F 95 80 33 50
000189 Joel 19 M 70 98 55 72
000152 Bill 17 M 84 79 85 62
```

```
fsed ‘s/0/-/1’ SCORES
---149 Lyn 18 F 95 80 33 50
---189 Joel 19 M 70 98 55 72
---152 Bill 17 M 84 79 85 62
```

```
fsed -e ‘s/[0-9]/*/3’ SCORES
000149 Lyn ** F 95 80 33 50
000189 Joel ** M 70 98 55 72
000152 Bill ** M 84 79 85 62
```

keycut

Divides a file into sub-files according to the specified key. These sub-files will contain only the records with the chosen key.

keycut %<K_1> %<K_2> … %<K_N> <file>

US_DATA

```
01 Massachusetts 03 Springfield  82 0  23 84 10
01 Massachusetts 01 Boston       91 59 20 76 54
02 New_York      04 Manhattan    30 50 71 36 30
02 New_York      05 Brooklyn     78 13 44 28 51
03 New_Jersey    10 Newark       52 91 44 9  0
03 New_Jersey    12 Moorestown   95 60 35 93 76
04 Pennsylvania  13 Philadelphia 92 56 83 96 75
04 Pennsylvania  16 Hershey      45 21 24 39 03
```

```
keycut STATE_DATA%1 US_DATA
```

STATE_DATA.01

```
01 Massachusetts 03 Springfield  82 0  23 84 10
01 Massachusetts 01 Boston       91 59 20 76 54
```

STATE_DATA.03

```
03 New_Jersey    10 Newark       52 91 44 9  0
03 New_Jersey    12 Moorestown   95 60 35 93 76
```

STATE_DATA.02

```
02 New_York      04 Manhattan    30 50 71 36 30
02 New_York      05 Brooklyn     78 13 44 28 51
```

STATE_DATA.04

```
04 Pennsylvania  13 Philadelphia 92 56 83 96 75
04 Pennsylvania  16 Hershey      45 21 24 39 03
```

dayslash

Transformation of Time/Date Format.

With the ***--output*** option, the existing date will be converted to a chosen format. With the ***--input*** option, the existing date will be converted from its current format to the default format (yyyymmdd).

dayslash [option] <date_format> <field_1> <field_2> … <field_N> <file>

DATES

20210101 2021/02/01 20210301

20210102 2021/02/02 20210302

20210103 2021/02/03 20210303

dayslash ***--output*** yyyy/mm/dd 1 3 DATES

2021/01/01 2021/02/01 2021/03/01
2021/01/02 2021/02/02 2021/03/02
2021/01/03 2021/02/03 2021/03/03

dayslash ***--input*** yyyy/mm/dd 2 DATES

20210101 20210201 20210301
20210102 20210202 20210302
20210103 20210203 20210303

comma

Adds comma as numeric separator.

comma <field_1> <field_2> … <field_N> <file>

SAMPLE_NUMS

1234567890

98765

13243546

comma 1 SAMPLE_NUMS

1,234,567,890
98,765
13,243,546

rjson

Parses a json file removing the structural tags and converting it to a tabular format.

rjson <json_file>

SAMPLE.json

```
[
  {
    "journal": "Sample Journal",
    "category": [
      "Examples",
      "Text Examples",
      "rjson showcase"
    ],
    "year": "2022",
    "journalText": "This is a sample text from the Sample Journal."
    "id": "A0001",
    "title": "Sample title"
]
```

```
rjson SAMPLE.json

1 journal Sample_Journal
1 category 1 Examples
1 category 2 Text_Examples
1 category 3 rjson_showcase
1 year 2022
1 journalText This_is_a_sample_text_from_the_Sample_Journal.
1 id A0001
1 title Sample_title
```

xmldir

Parses an xml file based on a given path, removing the original structure, and converting it to a tabular format. The ***-c*** option adds index fields based on the given number ***n***, and the specified xml path.

xmldir [-c<n>] /<DirTag1>/<DirTag2>/.../<DirTagN> <xml_file>

SAMPLE_1.xml

```
<dir1 id="A">
  <dir2 id="B">
    <day>23/Jul.2015</day>
    <day>24/Jul.2015</day>
  </dir2>
  <dir2 id="C">
    <day>25/Jul.2015</day>
    <day>26/Jul.2015</day>
  </dir2>
</dir1>
```

```
xmldir /dir1/dir2 SAMPLE_1.xml

dir1 id A dir2 id B day 23/Jul.2015
dir1 id A dir2 id B day 24/Jul.2015
dir1 id A dir2 id C day 25/Jul.2015
dir1 id A dir2 id C day 26/Jul.2015


xmldir -c3 /dir1/dir2 SAMPLE_1.xml
001 001 dir1 id A dir2 id B day 23/Jul.2015
001 001 dir1 id A dir2 id B day 24/Jul.2015
001 002 dir1 id A dir2 id C day 25/Jul.2015
001 002 dir1 id A dir2 id C day 26/Jul.2015
```

SAMPLE_2.xml

```
<dir1>
    <dir2 id="0">
        <attributes>
            <dir4 id="X">
                <header1>xxxx</header1>
                <header2>yyyy</header2>
            </dir4 id="X">
        </attributes>
    </dir2>
    <dir2 id="1">
        <attributes>
            <data>a</data>
            <data>b</data>
        </attributes>
        <attributes>
            <data>c</data>
        </attributes>
    </dir2>
</dir1>
<dir1>
    <dir2 id="2">
        <attributes>
            <data>d</data>
        </attributes>
        <attributes>
            <data>e</data>
        </attributes>
        <attributes>
            <data>f</data>
        </attributes>
    </dir2>
</dir1>
```

```
xmldir -c3 /dir1/dir2/attributes SAMPLE_2.xml
001 001 001 dir1 dir2 id 0 attributes dir4 header1 xxxx
001 001 001 dir1 dir2 id 0 attributes dir4 header2 yyyy
001 002 001 dir1 dir2 id 1 attributes data a
001 002 001 dir1 dir2 id 1 attributes data b
001 002 002 dir1 dir2 id 1 attributes data c
002 001 001 dir1 dir2 id 2 attributes data d
002 001 002 dir1 dir2 id 2 attributes data e
002 001 003 dir1 dir2 id 2 attributes data f
```

4. USE CASES

In this section, we present some example use cases were Unicage can be used as powerful processing tool. **Note that these examples might require the usage of commands that are only present in the Unicage Enterprise Version.**

Reporting from CDR Data

In order to keep track of the usage of their infrastructure, telecommunications companies rely on Call Detail Records (CDRs). These are standardized records that are created to document every exchange a given communication device makes with the network. CDR data sits at the base of the billing calculations, which means that its analysis is a fundamental element in the Telco industry.

A CDR contains a wealth of information that characterizes the usage of a network by its users. For instance, a phone call generates a CDR that contains the phone numbers involved in the call, the time, the date, the duration of the call, among other information.

Let us take as an example the file CDR_SAMPLE.csv, a file where each line is composed by 42 fields:

```
$ head -1 CDR_SAMPLES.csv

"G" ,"0" ," +351999990665 " ," +351999994370 " ," 12/02/2020 " ," 01:00:00 " ," 2865 " ," 837384086 "
,"206787737 " ," Faro " ,"" ," Special3 " ," .2577 " ," 1221 " ," +351999990321 " ," 2747 " ," BT312CR " ,"
Name " ,"1" ,"E" ," PT " ," NET2 " ," " ,"" ," +351999996172 " ," 719 " ,"" ," EUR " ,"" ," " ,"" ," " ,"
1044171823 " ,"userID52260 " ,"" ," " ," 361464 " ,"" ," "
```

For this use case, we will focus on a single objective: Make a report of the calls made by each Portuguese phone number, presenting the date and time, the dialled number, the total number of minutes of the call and a percentage of the time with respect to the total duration of the calls made by that number.

The first step in our process will be converting the original .csv file to a tabular text file, i.e., “a space-separated value” file. This can be done with the Unicage command **fromcsv**:

```
$ fromcsv CDR_SAMPLES.csv > CDR_SAMPLES_TABULAR

$ head -1 CDR_SAMPLES_TABULAR
G 0 +351999990665 +351999994370 12/02/2020 01:00:00 2865 837384086 206787737 Faro _ Special3
.2577 1221 +351999990321 2747 BT312CR Name 1 E PT NET2 _ _ _ +351999996172 719 _ EUR _ _ _ _
1044171823 userID52260 _ _ 361464 _ _
```

As we can see, the data is the same, but all the quotation marks are gone. In fact, only the relevant data is kept (i.e., the actual information regarding the phone calls) while the unnecessary data is left behind (the quotation marks). Additionally, the file size will decrease when compared with the original .csv file. This is a crucial step in Unicage data processing since we can save storage space by removing unnecessary data.

The next step will focus on the calculations that we want to make. For this we will build a pipeline divided in three steps:

1. Isolate the relevant data.
2. Guarantee the quality of the data.
3. Sort the data.

To isolate the relevant CDR data, we will use the Unicage **self** command to select the fields "CustomerNumber", "CallDate", "CallTime", "NumberDialled" and "Duration". These fields are at positions 3, 5, 6, 4 and 7, respectively, in the CDR file.

```
self 3 5 6 4 7 CDR_SAMPLES_TABULAR
```

Next, we will use Unix's awk command to implement a data quality filter that fills the following requirements:

- The phone numbers need to have exactly 13 characters, including the "+".
- The 1st character must be "+" and from the 2nd to the 13th character we have digits (the actual number including the country code)
- The 2nd, 3rd and 4th digit need to correspond to the country code for Portugal, which is 351.

These requirements can be fulfilled with the following awk pipeline:

```
awk 'length ($1) == 13'          |
awk '$1 ~ /^+[0 -9]{12}/ '       |
awk 'substr ($1 ,2 ,3) == "351"'  |
```

Then we repeat this pipeline, but for the field "NumberDialled", which will correspond to the 4th field.

```
awk 'length ($4) == 13'          |
awk '$1 ~ /^+[0 -9]{12}/ '       |
awk 'substr ($4 ,2 ,3) == "351"'  |
```

Finally, we finish this pipeline by sorting the data by phone number using Unicage's **msort** command. This way, we shall obtain a file with the records of each phone call made, along with the total time.

```
msort key=1 > tmp_phone_calls_duration
```

And the resulting file, "tmp_phone_calls_duration", is a tabular text file with the following format:

```
$ head -5 tmp_phone_calls_duration
+351999990000 12/02/2020 01:00:00 +351999994375 3558
+351999990000 12/02/2020 01:00:00 +351999991637 1076
+351999990000 12/02/2020 01:00:00 +351999990235 1271
+351999990000 12/02/2020 01:00:00 +351999993094 1804
+351999990000 12/02/2020 01:00:00 +351999990902 102
```

The next step shall be summing the duration of the calls made by each number. This is easily done by taking the "tmp_phone_calls_duration" file and using Unicage's **sm2** command, which sums the values of a given field or fields for each record with the same key.

```
$ sm2 1 1 5 5 tmp_phone_calls_duration  > tmp_number_total_duration

$ head -5 tmp_number_total_duration
+351999990000 4101500
+351999990001 3427680
+351999990002 3873560
+351999990003 4553480
+351999990004 3258580
```

Now, we want to add to each record in the file "tmp_phone_calls_duration", the total call duration of each phone number that is present in the file "tmp_number_total_duration". To achieve this, we use Unicage's **join2** command to add the information of the latter file to the information of tmp_phone_calls_duration:

```
$ join2 key =1 tmp_number_total_duration tmp_phone_calls_duration
```

The resulting output will contain the information that was added from the "tmp_number_total_duration" (in red for illustrative purposes), and add it to the "tmp_phone_calls_duration":

```
+351999990000 4101500 12/02/2020 01:00:00 +351999994375 3558
+351999990000 4101500 12/02/2020 01:00:00 +351999991637 1076
+351999990000 4101500 12/02/2020 01:00:00 +351999990235 1271
+351999990000 4101500 12/02/2020 01:00:00 +351999993094 1804
+351999990000 4101500 12/02/2020 01:00:00 +351999990902 102
```

With all this information present in a single file, we can calculate the percentage of time a given phone call took with respect to the total duration of calls. To do this, we use Unicage's **lcalc** command. This command allows the user to make calculations and attain precise results. So, to calculate the percentage, we need to divide the Call Duration (6th field) by the Total Duration of Calls made by the number (2nd field) and multiply the resulting value by 100. The formula is highlighted in red:

```
$ lcalc '$1 , $3 , $4 , $5 , $6 , $6/$2 *100 '
```

The resulting output will be the following, with the percentage highlighted in red:

```
+351999990000 12/02/2020 01:00:00 +351999994375 3558 0.086748750457149800
+351999990000 12/02/2020 01:00:00 +351999991637 1076 0.026234304522735500
+351999990000 12/02/2020 01:00:00 +351999990235 1271 0.030988662684383700
+351999990000 12/02/2020 01:00:00 +351999993094 1804 0.043983908326222100
+351999990000 12/02/2020 01:00:00 +351999990902 102 0.002486895038400500
```

Now, we just need to use Unicage's **round** command to get better presentation numbers for the percentage field before saving the final file. We choose to use 3 decimal places:

```
$ round 6.3 > FINAL_RESULT

$ head -5 FINAL_RESULT
+351999990000 12/02/2020 01:00:00 +351999994375 3558 0.087
+351999990000 12/02/2020 01:00:00 +351999991637 1076 0.026
+351999990000 12/02/2020 01:00:00 +351999990235 1271 0.031
+351999990000 12/02/2020 01:00:00 +351999993094 1804 0.044
+351999990000 12/02/2020 01:00:00 +351999990902 102 0.002
```

All the commands and steps described in this use case can be combined in a single script, that should look like this (code commentaries are in blue):

```
# convert from .csv to Unicage Format
# ----------------------------------
fromcsv CDR_SAMPLES.csv > CDR_SAMPLES_TABULAR
# ----------------------------------
# 1. Isolating the data needed for our analysis
self 3 5 6 4 7 CDR_SAMPLES_TABULAR |
# 1: CustomerIdentifier 2: CallDate 3: CallTime 4: NumberDialled 5: Duration
# ----------------------------------
# 2. Applying filters to the phone number making the call
awk 'length ($1) == 13' |
awk '$1 ~ /^+[0 -9]{12}/ ' |
awk 'substr ($1 ,2 ,3) == "351"' |
# ----------------------------------
# 3. Applying filters to the phone number receiving the call
awk 'length ($4) == 13' |
awk '$4 ~ /^+[0 -9]{12}/ ' |
awk 'substr ($4 ,2 ,3) == "351"' |
# ----------------------------------
# 4. sorting by phone number tha made the call
msort key =1 > tmp_phone_calls_duration
# 1: CustomerIdentifier 2: CallDate 3: CallTime 4: NumberDialled 5: Duration
```

```
self 1 5 tmp_phone_calls_duration |
# 1: CustomerIdentifier 2: Duration
# ---------------------------------
# 4. calculate the total " call duration " for each number
sm2 1 1 2 2 > tmp_number_total_duration
# 1: CustomerIdentifier 2: Duration ( total )

# --- 4. joining the total durantion for each number
join2 key =1 tmp_number_total_duration tmp_phone_calls_duration |
# 1: CustomerIdentifier 2: Duration ( total ) 3: CallDate 4: CallTime 5: NumberDialled 6: Duration
# ---------------------------------
# --- 5. Calculating the percentage of duration of each call with
# --- respect to the total durantion of each client
lcalc '$1 , $3 , $4 , $5 , $6 , $6/$2 *100 ' |
# 1: CustomerIdentifier 2: CallDate 3: CallTime 4: NumberDialled 5: Duration 6: Duration (
percentage )
# ---------------------------------
# --- 6. Rounding percentage to two decimal places
round 6.3 > FINAL RESULT
```

Combining and normalizing Sensor data types

Taming data in different formats is a common challenge in any data pipeline. In what follows, we will consider a scenario in which such a challenge occurs: combining data coming from different types of sensors. The sensors come from different manufacturers that follow different standards, which demands the inclusion of a normalization procedure. This procedure must reconcile all the data formats, so that all the information potential can be extracted from the data.

For this use case, our objective is going to be using the Unicage tools to build a data pipeline that is able to normalize two different data formats and combine them in a ready-to-plot csv file.

To start, let us assume that we have two files that we receive from two different models of vending machines: one model produces json files and the other produces xml files. Each file contains information about the items that are delivered by each vending machine. Every item that is delivered represents an event and generates packet of data that includes the code that identifies the item, the date and time at which the item was delivered, the ID of the machine that delivered the item and the shelf from where the item was taken (upper, middle, or lower). This information is organized in the files as follows:

```
$ head -15 model01_file.json
{
"1": {
"prateleira": "superior",
"produto": "34WE4",
"maquina": "C010",
"hora": "16:29",
"data": "24-01-2021"
}
},{
"2": {
"prateleira": "meio",
"produto": "21WX5",
```

```
$ head -15 model02_file.xml
<events>
<event>
<date>2021-04-21</date>
<time>11:05</time>
<equipment>C012</equipment>
<item>87WV2</item>
<shelf>upper</shelf>
</event>
<event>
<date>2021-04-21</date>
<time>03:23</time>
<equipment>C014</equipment>
```

As we can see, other than the file format, the information that is present is in different languages and the date format is also different. To tackle this data according to our goal, we must normalize both files to satisfy the following concerns:

- The final csv file must have a header describing the data fields written in English.
- The date fields must be presented in the YYYY/MM/DD format.
- The values of the “shelf” field must be written in English.

We will start by converting the json file to tabular text file. This is a fairly simple process with Unicage’s **rjson** command that parses the json file and converts it to a text file:

```
$ rjson model01_file.json
1 prateleira superior
1 produto 34EW4
1 maquina C010
1 hora 16:29
1 data 24-01-2021
2 prateleira meio
2 produto 21WX5
```

Looking at the generated output, we can see that the number of the event is in the first column (blue), the name of the data field is in the second column and the value of the data field in the third (red). With the data in this format, we can proceed to the task of translating the field's names. We can save the corresponding translations between the words in Portuguese and English in a simple text file ("translation_types"), with the English word on the 1st column and the corresponding Portuguese word on the 2nd column:

```
$ cat translations_types
data date
hora time
maquina equipment
prateleira shelf
produto item
```

Then, we can use this file in conjunction with the data in tabular format from the json file as inputs to Unicage's **cjoin2** command, like this:

```
$ rjson model01_file.json | cjoin2 key=2 translations_type -
1 prateleira shelf superior
1 produto item 34WE4
1 maquina equipment C010
1 hora time 16:29
1 data date 24-01-2021
2 prateleira shelf meio
2 produto item 58W77
```

As we can see, in front of each Portuguese word, the corresponding English word was added (green). We can then do the same but for the values of the field "shelf"/"prateleira".

```
$ cat translations_field
inferior lower
meio middle
superior upper

$ rjson model01_file.json | cjoin2 key=2 translations_type - | cjoin2 +? key=4 translations_field -
1 prateleira shelf superior upper
1 produto item 34WE4 ?
1 maquina equipment C010 ?
1 hora time 16:29 ?
1 data date 24-01-2021 ?
2 prateleira shelf meio middle
2 produto item 58W77 ?
```

With this, the rightmost field of the output contains the translation of the value of the field "shelf"/"prateleira" in the corresponding lines and "?" in all the other lines. To complete this process, we just need to select the relevant fields, by using both Unicage's **self** command and awk to remove the "?" characters from the lines and normalize the data.

```
$ rjson model01_file.json | cjoin2 key=2 translations_type - |
cjoin2 +? key=4 translations_field - |
self 1 3 4 5 | awk '{if($4 != "?"){$3 = $4}; print $1, $2, $3}' > converted_json_UF

$ head -7 converted_json_UF
1 shelf upper
1 item 34WE4
1 equipment C010
1 time 16:29
1 date 24-01-2021
2 shelf middle
2 item 58W77
```

And thus, we have completed the conversion of the json file to a tabular format and we were able to translate some words from Portuguese to English. Now, we can start the normalization of the xml file. This can be done using Unicage's **xmldir** command. This command requires the specification of the hierarchy of the xml file, which is done with the string "events/event" highlighted in blue:

```
$ xmldir events/event model02_file.xml
events event date 2021-04-21
events event time 11:05
events event equipment C012
events event item 87WV2
events event shelf upper
events event date 2021-04-21
events event time 03:23
```

Notice that the first two columns are the hierarchy tags of the xml file. We are not interested in these first two columns, so we discard them with Unicage's **self** command

```
$ xmldir events/event model02_file.xml | self 3 4
date 2021-04-21
time 11:05
equipment C012
item 87WV2
shelf upper
date 2021-04-21
time 03:23
```

At this stage, the data is very similar to the data from the converted json file. The difference is that we don't have a number to identify each event. We can add the numbering of the events by noticing that the data fields associated to a given event are aggregated in groups of 5, i.e., from line 1 to line 5 we have data that correspond an event, from line 6 to line 10 we have data that corresponds to another event, and so on. To implement this logic, we use the awk command:

```
$ xmldir events/event model02_file.xml | self 3 4 |
awk 'BEGIN{event_number=1;event_counter=0}{if(event_counter == 5){event_counter =
0;event_number++}
else{event_counter++}; print event_number, $0}' > converted_xml_UF
$ head -7 converted_xml_UF
1 date 2021-04-21
1 time 11:05
1 equipment C012
1 item 87WV2
1 shelf upper
2 date 2021-04-21
2 time 03:23
```

And this way we finished the normalization of both files. The final step will be combining the two normalized files into a single .csv file. We will start by transposing the data with Unicage's command **map**:

```
$ map num=1 converted_json_UF
* date equipment item shelf time
1 24-01-2021 C010 34WE4 upper 16:29
2 24-01-2021 C024 58W77 middle 20:11
...

$ map num=1 converted_xml_UF
* date equipment item shelf time
1 2021-04-21 C012 87WV2 upper 11:05
2 2021-04-21 C014 32WI7 middle 03:23
...
```

Having the data in this format, we will harmonize the dates so that they conform to the requirement for the .csv report. We notice that the file "converted_xml_UF" has the date in the format YYYY-MM-DD, which is almost the same as the date format of the requirement. Since the file "converted_json_UF" has the date in the format DD-MM-YYYY, we will start by using Unicage's command **dayslash** to convert from one date format to the other. In addition, we can remove the header using tail -n+2 and can save the data in a temporary file named "temp-json_data":

```
$ map num=1 converted_json_UF | tail -n+2 | dayslash --input dd-mm-yyyy --output yyyy-mm-dd 2 > temp-json_data
$ head -2 temp-json_data
1 2021-01-24 C010 34WE4 upper 16:29
2 2021-01-24 C024 58W77 middle 20:11
```

Then, we will isolate the header of the file using the head command and we save the result in a temporary file named temp-header:

```
$ map num=1 converted_json_UF | head -1 > temp-header

$ cat temp-header
* date equipment item shelf time
```

Now, we can go back to the file "converted_xml_UF" and prepare it to be concatenated with "temp-json_data" by removing the header. The concatenation is made with the cat command:

```
$ map num=1 converted_xml_UF | tail -n+2 |
cat temp-json_data -
1 2021-04-21 C012 87WV2 upper 11:05
2 2021-04-21 C014 32WI7 middle 03:23
...
1 2021-01-24 C010 34WE4 upper 16:29
2 2021-01-24 C024 58W77 middle 20:11
...
```

This command leaves the data ready for the last operation on the date field: we will use Unicage's **dayslash** command to convert the date from YYYY-MM-DD to YYYY/MM/DD.

```
$ map num=1 converted_xml_UF | tail -n+2 |
cat temp-json_data - |
dayslash --input yyyy-mm-dd --output yyyy/mm/dd 2
1 2021/01/24 C010 34WE4 upper 16:29
2 2021/01/24 C024 58W77 middle 20:11
...
1 2021/04/21 C012 87WV2 upper 11:05
2 2021/04/21 C014 32WI7 middle 03:23
...
```

Finally, we concatenate the header and remove the field with the event number using Unicage's **self** command. Then, we convert the result to .csv format using Unicage's **tocsv** command, a useful command to create .csv files.

```
$ map num=1 converted_xml_UF | tail -n+2 |
cat temp-json_data - |
dayslash --input yyyy-mm-dd --output yyyy/mm/dd 2 |
cat temp-header - |
self 2 3 4 5 6 |
tocsv  > combined_data.csv
2021/01/24,C010,34WE4,upper,16:29
2021/01/24,C024,58W77,middle,20:11
...
2021/04/21,C012,87WV2,upper,11:05
2021/04/21,C014,32WI7,middle,03:23
...
```

All the steps and commands described can be combined in a single script that should look like this (code commentaries are in blue):

```
# **** PART 1 ****
# ================
# --- a) parse the json file
rjson model01_file.json                          |
# 1:event 2:field_name(portuguese) 3:field_value
# --- b) add the translation of the field names
cjoin2 key=2 translations_type -                 |
# 1:event 2:field_name(portuguese) 3:field_name(english) 4:field_value
# --- c) add the translation of values of one field
cjoin2 +? key=4 translations_field -             |
# 1:event 2:field_name(portuguese) 3:field_name(english) 4:field_value 5:translated_field_value
# --- d) discard the field names in portuguese
self 1 3 4 5                                     |
# 1:event 2:field_name(english) 3:field_value 4:translated_field_value
# --- e) select the values of "shelf" in english
awk '{if($4 != "?"){$3 = $4}; print $1, $2, $3}'       > converted_json_UF
# 1:event 2:field_name(english) 3:field_value


# **** PART 2 ****
# ================
# --- a) parse the xml file
xmldir events/event model02_file.xml             |
# 1:hierarchy_tag 2:hierarchy_tag 3:field_name 4:field_value
# --- b) discard the hierarchy tags
self 3 4                                         |
# 1:field_name 2:field_value
# --- c) adding an event number to the data
awk 'BEGIN{event_number=1;event_counter=0}
        {if(event_counter == 5){event_counter = 1;event_number++}
         else{event_counter++};
         print event_number, $0}'                > converted_xml_UF
# 1:event 2:field_name 3:field_value
```

```
# **** PART 3 ****
# ================
# *** 3.1: preparing the json data
# --- a) transposing the json data
map num=1 converted_json_UF              |
# 1:event 2:date(DD-MM-YYYY) 3:equipment 4:item 5:shelf 6:time
# --- b) discarding the header
tail -n+2                    |
# 1:event 2:date(DD-MM-YYYY) 3:equipment 4:item 5:shelf 6:time
# --- c) converting the date format
dayslash --input dd-mm-yyyy --output yyyy-mm-dd 2   > temp-json_data
# 1:event 2:date(YYYY-MM-DD) 3:equipment 4:item 5:shelf 6:time

# *** 3.2: separating the header
# --- a) transposing the json data
map num=1 converted_json_UF              |
# 1:event 2:date 3:equipment 4:item 5:shelf 6:time
# --- b) isolating the header
head -1                      > temp-header
# 1:event 2:date 3:equipment 4:item 5:shelf 6:time

# *** 3.3: combining all the data and producing the csv file
# --- a) transposing the xml data
map num=1 converted_xml_UF           |
# 1:event 2:date(YYYY-MM-DD) 3:equipment 4:item 5:shelf 6:time
# --- b) discarding the header
tail -n+2                    |
# 1:event 2:date(YYYY-MM-DD) 3:equipment 4:item 5:shelf 6:time
# --- c) combining with the json data in the temporary file
cat temp-json_data -                 |
# 1:event 2:date(YYYY-MM-DD) 3:equipment 4:item 5:shelf 6:time
# --- d) converting the format of the date field
dayslash --input yyyy-mm-dd --output yyyy/mm/dd 2    |
# 1:event 2:date(YYYY/MM/DD) 3:equipment 4:item 5:shelf 6:time
# --- e) concatenating the header
cat temp-header -                |
# 1:event 2:date(YYYY/MM/DD) 3:equipment 4:item 5:shelf 6:time
# --- f) discarding the field "event"
self 2 3 4 5 6                   |
# 1:date(YYYY/MM/DD) 2:equipment 3:item 4:shelf 5:time
# --- g) converting to csv format
tocsv                        > combined_data.csv
# date(YYYY/MM/DD),equipment,item,shelf,time
```

Smart Meters

Smart meters are devices that record and communicate information regarding the consumption of energy or other resources. Having smart meters in place allows the energy grid to automatically respond to conditions and events within it, which can lead to optimized energy distribution. However, for this to be possible, it is crucial to gather and disseminate data with strict time requirements. This creates the need for a system that can process and manage large amounts of data efficiently. The amount of data generated by smart meters depends on the frequency with which readings from the smart meters are collected.

In this case, we shall process data received from 100.000 smart meters. Each smart meter produces an XML file containing the different types of readings. An example is shown below:

```
$ cat SM000000001VG_20210101T002650.xml
<MeterReadings>
<MeterReading>
<Meter>
<Names>
<name>SM000000001VG</name>
<NameType>
<description>This is a meter identification number.</description>
<name>MeterID</name>
</NameType>
</Names>
</Meter>
<Readings>
<timeStamp>2021-01-01T00:26:50Z</timeStamp>
<value>12.6316</value>
<ReadingType ref="0.0.0.4.1.1.12.0.0.0.0.0.0.0.3.72.0"/>
</Readings>
<Readings>
<timeStamp>2021-01-01T00:26:50Z</timeStamp>
<value>11.8440</value>
<ReadingType ref="0.0.0.12.1.1.37.0.0.0.0.0.0.0.3.38.0"/>
</Readings>
<Readings>
<timeStamp>2021-01-01T00:26:50Z</timeStamp>
<value>6.5043</value>
<ReadingType ref="0.0.0.0.0.0.46.0.0.0.0.0.0.0.0.23.0"/>
</Readings>
</MeterReading>
</MeterReadings>
```

For this use case, our objective will be processing the data from the 100.000 smart meters, make a simple validation of the readings and create a final file containing the sum of the values from all valid readings. To achieve this, since the files are in xml format, we start by using Unicage's **xmldir** command to parse

the 100.000 files and then merge this data into a single file. For this, we use a combination of Unicage's **self** and **delr** commands together with awk to prepare the data so that it can be transposed using Unicage's **map** command:

```
find -name "*.xml" -exec cat {} + |
xmldir -c3 /MeterReadings/MeterReading - |
self NF-1 NF |
awk '$1 == "name" || $1 == "timeStamp" || $1 == "value" || $1 == "ref"' |
delr 2 "MeterID" |
awk '{if($1=="name"){meter=$0;print $0}
        else{
           if($1 == "timeStamp"){print meter; print $0}
           else{print $0}
           }
       }'              |
awk '{if($1 == "name"){count++}; print count, $0}' |
map num=1 |
delf 1 |
delr 3 "0" > READINGS_PARSED
```

The output will be a single file named "READINGS_PARSED" which will have the following structure, with the smart meter id in the 1st column (red), the reading reference in the 2nd (blue), the timestamp in the 3rd and the reading value in the 4th column (green):

```
SM000000689VG 0.0.0.4.1.1.12.0.0.0.0.0.0.0.3.72.0 2021-01-01T12:40:06Z 14.8361
SM000000689VG 0.0.0.12.1.1.37.0.0.0.0.0.0.0.3.38.0 2021-01-01T12:40:06Z 7.4433
SM000000689VG 0.0.0.0.0.46.0.0.0.0.0.0.0.0.23.0 2021-01-01T12:40:06Z 6.5668
SM000000145VG 0.0.0.4.1.1.12.0.0.0.0.0.0.0.3.72.0 2021-01-01T08:54:15Z 19.7668
SM000000145VG 0.0.0.12.1.1.37.0.0.0.0.0.0.0.3.38.0 2021-01-01T08:54:15Z 10.1405
SM000000145VG 0.0.0.0.0.46.0.0.0.0.0.0.0.0.23.0 2021-01-01T08:54:15Z 6.9721
SM000000453VG 0.0.0.4.1.1.12.0.0.0.0.0.0.0.3.72.0 2021-01-01T06:50:54Z 9.9979
SM000000453VG 0.0.0.12.1.1.37.0.0.0.0.0.0.0.3.38.0 2021-01-01T06:50:54Z 19.0457
SM000000453VG 0.0.0.0.0.46.0.0.0.0.0.0.0.0.23.0 2021-01-01T06:50:54Z 14.0774
(…)
```

The next step will focus on the validation of the readings. For this, we have a Master file called "READING_TYPE_CONVERTER" with the reading types that are considered valid. In addition, we have a "TYPE0X" that is associated with each type of reading. This "TYPE0X" can represent, for example, Electricity, Gas or Water, depending on what the smart meter is measuring:

```
$ cat READING_TYPE_CONVERTER
0.0.0.0.0.46.0.0.0.0.0.0.0.0.23.0 TYPE01
0.0.0.12.1.1.37.0.0.0.0.0.0.0.3.38.0 TYPE02
0.0.0.4.1.1.12.0.0.0.0.0.0.0.3.72.0 TYPE03
```

With this Master file, we can then use Unicage's **cjoin1** command to identify which readings are valid, whilst adding the corresponding type to the reading. To remove the readings which do not appear in the Master file, we use the *+ng* option and redirect its output to a file which will contain all the invalid readings that are present in the merged smart meter file.

```
cat  READINGS_PARSED  |
cjoin1 +ng3 key=2 READING_TYPE_CONVERTER -  > ALL_VALID_READINGS   3>  ALL_INVALID_READINGS
```

With the readings split between valid and invalid, we can advance to the final step, which is making the cumulative sum of the valid readings by their type. This is easily achieved by selecting the desired fields using Unicage's **self**, sort them using **msort** and, finally, sum all the values by using the **sm2** command:

```
self 3 5 ALL_VALID_READINGS    |
# 1:reading_type_name 2:value
msort key=1                                          |
# 1:reading_type_name 2:value
sm2 1 1 2 2   >  MEASURED_VALUES_by_TYPE
```

The final result will be the following file with the three types of valid readings and the respective cumulative value of all readings of that type present in the smart meter files:

```
$ cat MEASURED_VALUES_by_TYPE
TYPE01 10143.4150
TYPE02 9915.1172
TYPE03 10087.8375
```

Financial Reporting in XML and XBRL

The communication between financial regulators and institutions, such as banks and central banks, is crucial for the world's economy. For the communication between these entities to be efficient, standard file formats are used in order to encapsulate all information that is needed. Generally, these formats are XML and XBRL.

For both formats, our objective in this use case will be parsing the files to a tabular format, while we perform certain technical validations in order to split between valid and invalid data.

We shall divide this use case in two sections, one for each data format.

XML Financial Reporting

In this example, we use a file that mimics the report of a financial institution to a National Bank. The first 20 lines of our file are shown below:

```
$ head -20 example_report.xml
<IPSYS_PSP>
<Header>
<period>1933-04</period>
<PSP_reporter>AAAA</PSP_reporter>
<PSP_reported>AAAA</PSP_reported>
<PSP_destiny>0000</PSP_destiny>
<versionXSD>PT2.1.00</versionXSD>
<coment>...</coment>
<Instrument>
</Instrument>
<Checks>
<Ch_Draws>
<ChS_Details>
<chs_id>CHS.000000000a</chs_id>
<scheme></scheme>
<processors>1</processors>
<institutional_sector>S3</institutional_sector>
<check_type>9</check_type>
<operationg_country>PT</operationg_country>
<operating_division>EUR</operating_division>
```

As we can see, this xml file is separated in different sections that correspond to different types of elements to be reported. For instance, Checks, Transfers, etc. Having this information in a single xml file is very convenient for the communications phase, where the institution sends the information over a network to the National Bank, but the analysis phase becomes too burdensome. To simplify the analysis phase, one can parse the xml file and organize that data by instance in a tabular file.

During this process of parsing and organization, we can include a validation phase, where we can identify missing fields and mismatched data types. After performing this processing, the data can be more easily manipulated by visualization tools or other software packages that are used to post-process the data. By accomplishing this, not only we are able to turn a very complicated and hard to read file into several parsed and validated files, but we are also able to save about 60% of memory space by only getting the essential data of the file without any loss of quality.

We will start by replacing every missing field within the xml file with an "@". This is simply done using the built in sed tool:

```
$ cat example_report.xml | sed 's/></>@</g'  > example_report_mf.xml
```

Then, we will convert the data into a tabular text file. We will do this using Unicage's **xmldir** command in a loop to read each branch of the xml structure that interest us and that we have described inside an auxiliary file named "tmp-XML_PATH":

```
$ cat tmp-XML_PATH
/IPSYS_PSP/Instrument/Checks/Ch_Raised/ChS_Details/
/IPSYS_PSP/Instrument/Checks/Ch_Drawn/
/IPSYS_PSP/Instrument/AccountPayment/Account_Payment/
/IPSYS_PSP/Instrument/AccountPayment/Account_Movements/Mv_Details/
/IPSYS_PSP/Instrument/DirectDebits/
/IPSYS_PSP/Instrument/Efects/
/IPSYS_PSP/Instrument/MiddlePayments/PI_Record/
/IPSYS_PSP/Instrument/MiddlePayments/PI_Payment/
/IPSYS_PSP/Instrument/PaymentNetwork/RP_Creditor/RPC_Detail/RPC_Record/
/IPSYS_PSP/Instrument/PaymentNetwork/RP_Creditor/RPC_Detail/RPC_Payment/
/IPSYS_PSP/Instrument/PaymentNetwork/RP_Debter/RPD_Payment/
/IPSYS_PSP/Instrument/FraudReport/
/IPSYS_PSP/Instrument/PaymentTerminals/
/IPSYS_PSP/Instrument/Transfers/Tr_Emited/TrE_Details/
/IPSYS_PSP/Instrument/Transfers/Tr_Received/TrR_Details/
```

```
while read -r line; do
        # --- Parses file of the specific branch ---
        xmldir $(echo $line) example_report_mf.xml          |
        # --- Selects last 3 fields ---
        self NF-2 NF-1 NF                   |
        # --- Creates a numeric value field that increments from id to id, ---
        # --- which means from a full instance to another ---
        awk 'BEGIN{count=0}{if($2 ~ "^[a-zA-Z]+_id$"){count++}; print count, $0}'      >>
        TABULAR_FILES/$dirname/tmp-$filename
done < "tmp-XML_PATH"
```

After executing the code, the first 10 lines of our files, in this case, the “Checks” file, will have a structure similar to the one that is shown below:

```
1 ChS_Details chs_id CHS.000000000a
1 ChS_Details scheme @
1 ChS_Details processors 1
1 ChS_Details institutional_sector S3
1 ChS_Details check_type 9
1 ChS_Details operation_country PT
1 ChS_Details operation_division EUR
1 Operations quantity 8988
1 Operations amount 15412.53
1 Operations_Dev quantity 587
```

The next step will be processing these files so that we deal with some specific cases. In addition, we use Unicage’s **map** command to re-format the data file into a matrix consisting of row key fields, column key fields and the rest of the fields as data fields.

```
cat TABULAR_FILES/$dirname/tmp-$filename  |
# --- Deals with an exception on specific fields called,
Operacoes, Imp_Legal, Operacoes_Dev, Dev_Operacoes_Dev,
due to different level in deepness of the tree
we use this command to artificially say
--- this level is (N-1) ---
awk '{if($2=="Operacoes"){print $1,$2,"Operacoes_"$3,$4}
else if($2=="Imp_Legal"){print $1,$2,"ImpLegal_"$3,$4}
else if($2=="Operacoes_Dev"){print $1,$2,"OperacoesDev_"$3,$4}
else if($2=="Dev_Operacoes_Dev"){print $1,$2,"DevOperacoesDev_"$3,$4}
else{print $0}}'          |
# --- Removes the 2nd field ---
delf 2                    |
# --- Maps file to tabular format ---
map -m@ num=1             |
# --- Removes 1st field ---
delf 1            |
fcols --           > TABULAR_FILES/$dirname/$filename
```

An example of the output is shown below, we data stored in a much more readable format:

divisa_operacao	pais_operacao	processador	scheme	setor_institucional	tipo_transferencia	trr_id
EUR	X	1	1	S7	0	TRR.0000000001
EUR	X	1	1	S6	0	TRR.0000000002
EUR	X	1	1	S2	0	TRR.0000000003
EUR	X	1	1	S5	0	TRR.0000000004
EUR	X	1	1	S2	0	TRR.0000000005
EUR	X	1	1	S6	0	TRR.0000000006
EUR	X	1	1	S6	0	TRR.0000000007
EUR	X	1	1	S5	0	TRR.0000000008
EUR	X	1	1	S2	0	TRR.0000000009

The final step of the process is validating data for missing fields and mismatch datatypes. We shall take as an example the "Checks" file that is already in the tabular format. As we can see in the example bellow, we have a missing field that is represented by the "@" character highlighted in red:

DevOperacoesDev_montante	DevOperacoesDev_quantidade	ImpLegal_montante	ImpLegal_quantidade	OperacoesDev_montante	OperacoesDev_quantidade	Operacoes_montante	Operacoes_quantidade	chs_id	divisa_operacao	pais_operacao	processador	scheme	setor_institucional	tipo_cheque
8.42	898	7.77	7	1040.03	671	75191.47	30337	CHS.000000000a	EUR	PT	1	1	S7	9
2286.91	948	1.85	1	2357.33	801	44864.07	75483	CHS.0000000002	EUR	PT	1	1	S4	7
6890.53	878	14.47	2	3972.38	568	52403.68	@	CHS.0000000003	EUR	PT	1	1	S4	7
647.40	173	7.76	6	3414.78	726	302043.31	89854	CHS.0000000004	EUR	PT	1	1	S4	9
898.11	183	10.27	6	1929.11	448	479434.94	67094	CHS.0000000005	EUR	PT	1	1	S2	7
1282.50	382	21.54	3	10.19	10	242343.41	79813	CHS.0000000006	EUR	PT	1	1	S1	9
805.10	325	83.92	10	891.31	273	98729.25	11235	CHS.0000000007	EUR	PT	1	1	S5	3

To validate each field, we create a small script with validation criteria that the different fields must follow. To achieve this, we use a regular expression to instruct the grep command to exclude all the records that do not present the desired format:

```
# --- Reads file ---
cat $file                                                                    |
# --- Selects every field
self 1/NF                                                                    |
# --- Regular expression to catch every mismatch data type ---
grep -vP "^[0-9]*\.\d\d [0-9]* [0-9]*\.\d\d [0-9]* [0-9]*\.\d\d [0-9]* [0-9]*\.\d\d [0-9]* [A-Z]{3}\.[0-9]*
[A-Z]{2,4} [A-Z]{2,4} \d \d [A-Z][0-9] \d$"        |
fcols --                                                                                                  >
INVALID_DATA/$filename
```

The result is the file that we show bellow. As we can see, in addition to the record that we know that contained the missing field, there was also another record that did not pass the validation. This is the record with the value “CHS.000000000a” in the column “chs_id”:

```
DevOperacoesDev_montante DevOperacoesDev_quantidade ImpLegal_montante ImpLegal_quantidade
OperacoesDev_montante OperacoesDev_quantidade Operacoes_montante Operacoes_quantidade
chs_id       divisa_operacao pais_operacao processador scheme setor_institucional tipo_cheque
8.42              898                 7.77          7              1040.03          671              75191.47
30337          CHS.000000000a EUR          PT         1         1     S7             9
6890.53           878                 14.47         2              3972.38          568              52403.68
@              CHS.0000000003 EUR          PT         1         1     S4             7
```

XBRL Financial Reporting

In this example, we will focus on the XBRL reports used by the European Banking Authority (EBA). EBA uses a vast set of validation rules to check if a XBRL report is valid or not. However, this process is not trivial, not only due to the structure of the XBRL files, but also due to its contents which require specific validations. These validations are generally made by third party software that companies, banks and businesses use in order to check their XBRL reports. Taking this into account, we decided to choose a set of technical validation rules used by EBA and make a small demonstration to show that Unicage can also be a powerful tool to be used in XBRL validation and processing.

We are going to start by validating the namespaces used in the document. The namespaces are a technical header declared at the begin of the file and contains essential information for the whole report, from the XML schema information to information regarding the different taxonomies used within the report. Here is a small excerpt of this header:

```
<xbrli:xbrl xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xbrli="http://www.xbrl.org/2003/instance" xmlns:link="http://www.xbrl.org/2003/linkbase"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xbrldi="http://xbrl.org/2006/xbrldi"
xmlns:iso4217="http://www.xbrl.org/2003/iso4217"
xmlns:eba_model="http://www.eba.europa.eu/xbrl/ext/model"
(…)
```

With some small transformations using tr and in combination with Unicage's **makec** command, we can store the namespaces into a file where each record is a namespace. Then, after sorting each record with **msort**, we use **cjoin0** with the *+ng* option to save in a temporary file each namespace that is not present in a Master file called "EBA_NAMESPACES_ALL" that contains the list of valid namespaces for each type of report (for example, RESOLCON is a type of report):

```
# Retrieves only the namespace information line
tail -n+3 DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21.xbrl | head -n1 |
# Removes the '<' and '>' delimeters of the line
tr -d '<' | tr -d '>' |
# Removes '\r' if they exist
tr -d '\r' |
# Makes a record with each namespace
makec -1 num=1 |
# 1:xbrli_tag 2:namespace
delf 1 |
# 1:namespace
# Removes possible blank spaces
sed '/[[:blank:]]/d' |
# sorts and saves to tmp file
msort key=1 > tmp-NAMESPACES
```

```
report_type=RESOLCON

cat EBA_NAMESPACES_ALL |
selr 1 $report_type |
delf 1 |
cjoin0 +ng key=1 – tmp_NAMESPACES > /dev/null 2> tmp-NOK_NAMESPACES
```

Now, we check if the temporary file created is empty. If it is not empty, then there are invalid namespaces present, and the report is considered invalid. If the temporary file is empty, then the namespaces are valid, but we must check if the number of namespaces is equal to the number of namespaces present in the Master file. If it is different, then the file has less namespaces than required, thus it is invalid. To do this, we can use Unicage's **lcnt** command to in order to obtain the number of namespaces in each file.

```
namespaces_dim=$( selr 1 $report_type EBA_NAMESPACES_ALL | lcnt)
file_dim=$(lcnt DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21.xbrl)

if [ -s tmp-NOK_NAMESPACES ]
then
        echo “Invalid namespaces”
else
        if [ $file_dim != $namespaces_dim ]
        then
                echo “Invalid namespaces”
        fi
fi
```

The next step before we parse the full XBRL files is to validate the indicators. The indicators are the tables that are reported in the financial reports and each type of report has a group of characteristic tables. Thus, we need to validate if each reported table belongs to that type of report and if there are no repeated tables. Here is an example of the indicators section that is declared in a XBRL report:

```
 <find:fIndicators>
   <find:filingIndicator contextRef="c1">T_01.00</find:filingIndicator>
   <find:filingIndicator contextRef="c1">T_02.00</find:filingIndicator>
   <find:filingIndicator contextRef="c1">T_03.01</find:filingIndicator>
   <find:filingIndicator contextRef="c1">T_03.02</find:filingIndicator>
   <find:filingIndicator contextRef="c1">T_03.03</find:filingIndicator>
(…)
</find:fIndicators>
```

First, we need to parse the XBRL file extracting only the section that comprises the indicators. This can be easily made with Unicage's **xmldir** command since the XBRL language and structure is based on XML. Then, using **self**, we can select the indicators and store them in a temporary file for validation.

```
tail -n+3 DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21.xbrl |
# removes schema info for easy reading
sed -e 's/^<xbrli:xbrl.*/<xbrli:xbrl>/' |
# removes empty lines
sed '/^[[:space:]]*$/d' |
# fetches the indicators using xmldir
xmldir xbrli:xbrl/find:fIndicators - |
# 1-4:xbrl tags 5:id 6:indicator
self 6 > tmp-INDICATORS
```

In this validation, we check for repeated tables in the report through a combination of the uniq and Unicage's **lcnt** command:

```
# Checks if there are repeated values
if [ $(cat tmp-INDICATORS | uniq -d | lcnt) != 0 ]
```

If the same table is reported more than once, then the file is invalid. If there are no repeated values, then we follow the same rules as the namespace validation described earlier: we use Unicage's **cjoin0** with the *+ng* option to check if the reported tables for that type of report are correct according to a Master file containing the valid tables for each type of report.

After these initial validations, we can start parsing the entire XBRL file. Due to the similarities with XML, Unicage's **xmldir** command acts as the right tool to parse the XBRL file. However, since the files have plenty of definitions and declarations in the top hierarchy members, it is better to apply some sed commands to improve the readability before using **xmldir** to parse the entire file.

```
# PARSES XBRL FILE
# removes first 2 lines
tail -n+3 DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21_1.xbrl |
# removes schema info for easy reading
sed -e 's/^<xbrli:xbrl.*/<xbrli:xbrl>/' |
# removes empty lines
sed '/^[[:space:]]*$/d' |
# parsing using xmldir
xmldir xbrli:xbrl - >  PARSED_DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21_XBRL
```

The result will be a much more comprehensible file as seen in the following sample:

```
$ cat PARSED_DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21_XBRL
xbrli:xbrl link:schemaRef xlink:type simple xlink:href
http://www.eba.europa.eu/eu/fr/xbrl/crr/fws/res/cir-2018-1624/2021-07-15/mod/resol_con.xsd
xbrli:xbrl xbrli:unit xbrli:measure xbrli:pure
xbrli:xbrl xbrli:context xbrli:entity xbrli:identifier scheme http://standards.iso.org/iso/17442
DUMMYLEI123456789012
xbrli:xbrl xbrli:context xbrli:period xbrli:instant 2021-12-31
xbrli:xbrl find:fIndicators find:filingIndicator contextRef c1 T_01.00
xbrli:xbrl find:fIndicators find:filingIndicator contextRef c1 T_02.00
xbrli:xbrl find:fIndicators find:filingIndicator contextRef c1 T_03.01
xbrli:xbrl find:fIndicators find:filingIndicator contextRef c1 T_03.02
xbrli:xbrl find:fIndicators find:filingIndicator contextRef c1 T_03.03
(…)
xbrli:xbrl xbrli:context xbrli:entity xbrli:identifier scheme http://standards.iso.org/iso/17442
DUMMYLEI123456789012
xbrli:xbrl xbrli:context xbrli:period xbrli:instant 2021-12-31
xbrli:xbrl xbrli:context xbrli:scenario xbrldi:typedMember eba_typ:LE 1
xbrli:xbrl eba_met:si168 contextRef c2 8880
xbrli:xbrl eba_met:si288 contextRef c2 1883
xbrli:xbrl eba_met:ei555 contextRef c2 eba_ZZ:x428
xbrli:xbrl eba_met:ei152 contextRef c2 eba_GA:DK
xbrli:xbrl eba_met:bi628 contextRef c2 true
xbrli:xbrl eba_met:ei326 contextRef c2 eba_ZZ:x64
xbrli:xbrl eba_met:bi629 contextRef c2 true
xbrli:xbrl eba_met:ei4 contextRef c2 eba_AS:x2
xbrli:xbrl eba_met:bi630 contextRef c2 true
xbrli:xbrl xbrli:context xbrli:entity xbrli:identifier scheme http://standards.iso.org/iso/17442
DUMMYLEI123456789012
xbrli:xbrl xbrli:context xbrli:period xbrli:instant 2021-12-31
xbrli:xbrl xbrli:context xbrli:scenario xbrldi:typedMember eba_typ:LE 2
(…)
```

After the file has been parsed, we can perform other validations that are applied by EBA. In this case, we validate the identifier sequence of the XBRL file. This sequence must be equal across the entire file and comprises the link containing the XML schema, the file identifier, which usually is part of its name, and the instant, which corresponds to the data reference date. Here is an example of the identifier sequence in the original XBRL file and in the parsed file:

XBRL file:

```
 <xbrli:context id="c1">
  <xbrli:entity>
   <xbrli:identifier
scheme="http://standards.iso.org/iso/17442">DUMMYLEI123456789012</xbrli:identifier>
  </xbrli:entity>
  <xbrli:period>
   <xbrli:instant>2021-06-30</xbrli:instant>
  </xbrli:period>
 </xbrli:context>
```

Parsed file:

```
xbrli:xbrl xbrli:context xbrli:entity xbrli:identifier scheme http://standards.iso.org/iso/17442
DUMMYLEI123456789012
xbrli:xbrl xbrli:context xbrli:period xbrli:instant 2021-12-31
```

In order to extract the identifier sequence from the parsed XBRL file, we use a combination of grep and awk to extract only the lines that are of interest to us and organize them. Then, using Unicage's **self**, we can select only the fields of interest without the need of having all the other needless information, and store it in a temporary file for validation.

```
cat PARSED_DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21_XBRL |
# Retrieves only the lines containing the identifier sequence
grep -E '(xbrli:identifier|xbrli:instant)' |
# Uses Awk to join each 2 lines that comprise the identifier sequence
awk 'NR%2{printf "%s ",$0;next;}1' |
# 1-5: xbrl tags 6:schema 7:identifier 8-11:xbrl tags 12:instant
self 6 7 NF > tmp-IDENTIFIER-SEQUENCE
# 1:schema 2:identifier 3:instant
```

The result will be a temporary file containing all the identifier sequences used across the XBRL file:

```
$ cat tmp-IDENTIFIER-SEQUENCE
http://standards.iso.org/iso/17442 DUMMYLEI123456789012 2021-12-31
http://standards.iso.org/iso/17442 DUMMYLEI123456789012 2021-12-31
http://standards.iso.org/iso/17442 DUMMYLEI123456789012 2021-12-31
http://standards.iso.org/iso/17442 DUMMYLEI123456789012 2021-12-31
(…)
```

The first step in the validation of the identifier sequence is to check whether it is unique across the file. Through a combination of uniq and **lcnt** and check if the result is different than 1. If it is, that means that there is more than one identifier sequence and that the file is invalid. Then, if there is only one sequence, we need to validate if the date is a valid date and if the XML schema corresponds to a link. Both can be achieved with the usage of if conditions and some regular expressions.

```
if [ $(cat tmp-IDENTIFIER-SEQUENCE | delf 1 | uniq | lcnt) != 1 ]
then
        # if there are multiple identifier sequences across the file
        # sends the report to invalid and removes it from valid
        echo “Scheme Error: There are multiple identifiers”
else
        scheme=$(head -n1 tmp-IDENTIFIER-SEQUENCE | self 2 | uniq)
        instant=$(head -n1 tmp-IDENTIFIER-SEQUENCE | self 4 | uniq)
        val_link='(https?|ftp|file)://[-A-Za-z0-9\+&@#/%?=~_|!:,.;]*[-A-Za-z0-9\+&@#/%=~_|]'

        # Checks if Instant is a valid date
        if [[ $instant =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2}$ ]] && date -d "$instant" > /dev/null 2>&1
        then
                # if the Instant is a valid date, then it validates the scheme
                if [[ ! $scheme =~ $val_link ]]
                then
                # if the scheme is not a link, then it sends the report
                # to invalid and removes it from valid
                echo “Scheme Error: Link is invalid”
                fi
        else
                echo “Scheme Error: Date is invalid”
        fi
fi
```

If the identifier sequence is valid, we can advance to the extraction of the values corresponding to financial transactions and operations in Euros. To achieve this, we use the file corresponding to the parsed XBRL, from where we can use grep to extract the records that contain the monetary values and their corresponding information. In this case, we only care for values in Euros, so we use the Euro identifier that corresponds to ‘uEUR’ and store the information in a temporary file for further validation.

```
cat PARSED_DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21_XBRL |
# selects the lines that contain only the monetary values in Euros
grep ‘uEUR’ |
# 1:xbrli:xbrl 2:eba-tag 3:unitRef 4:uEUR 5:decimals-tag 6:decimals 7:contextRef 8:id 9:value
delf 1 > tmp-NUMERICS
# 1:eba-tag 2:unitRef 3:uEUR 4:decimals-tag 5:decimals 6:contextRef 7:id 8:value
```

After creating this file, we focus on two distinct validations: Validation of missing fields/data and validation of non-numeric values. For the first, we check for missing fields using Unicage's **ccnt**. We know that each correct record needs to have 8 columns. Therefore, if **ccnt** returns a value different than 8, this means that we are missing data, thus our report is invalid.

```
size=$(cat tmp-NUMERICS | ccnt)
# if the record size is not equal to 8, that means that there are missing values
if [[ $size != 8 ]];
then
        # There are missing fields
        echo “Missing/Empty Values"
fi
```

If we pass the previous validation, then we check for non-numeric values. To achieve it, we first use **self** to select only the field corresponding to the monetary value. Then, using grep, we store in a temporary file each value that is not a number. If the temporary file is not empty, then there are non-numeric values, thus the report is considered invalid. If the file is empty, then every value is numeric, and we can create the final file.

```
cat tmp-NUMERICS |
# 1:eba-tag 2:unitRef 3:uEUR 4:decimals-tag 5:decimals 6:contextRef 7:id 8:value
self 5 |
# 1:value
# selects each record that doesn't have a numeric value on it
# and stores it in a NIL tmp file
grep -v "[0-9]" > tmp-NIL

if [ -s tmp-NIL ];
then
# If the NIL tmp file is not empty, then it means that
# there are non-numeric values and the report is invalid
        echo “There are non-numeric values”
else
        # If the NIL tmp file is empty, then it means that
        # all values in the report are valid
        cat tmp-NUMERICS|
        # 1:eba-tag 2:unitRef 3:uEUR 4:decimals-tag 5:decimals 6:contextRef 7:id 8:value
        self 7 1 8 3 4 |
        # 1:id 2:eba-tag 3:value 4:uEUR 5:decimals
        awk 'BEGIN{print "id label/reason amount currency decimals"}1' |
        # adds header to file
        fcols > DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21-NUMERIC
fi
```

And thus, we finish this example of XBRL file processing. We reach the end with a file in a tabular format containing only the information regarding financial transactions in Euros, which can be easily used by other processing or visualization software without the need of further parsing or validations. Also, the information is much more comprehensible than in the original XBRL file:

```
$ cat DUMMYLEI123456789012_PT_RES010200_RESOLCON_2021-07-21-NUMERIC
id  reason amount currency decimals
c5  eba_met:mi53  2900000        uEUR      -3
c6  eba_met:mi53  9686000        uEUR      -3
c7  eba_met:mi53  6386000        uEUR      -3
c8 eba_met:mi235  9022000        uEUR      -3
c8 eba_met:mi236  7568000        uEUR      -3
(…)
```