EVL Anonymization Microservice Manual

EVL Tool products

- **EVL** (Extract–Validate–Load) – code based ETL tool (= Extract–Transform–Load);
- **EVL Worfklow** – code based orchestration tool;
- **EVL File Registration** – manipulate metadata about file processing;
- **EVL Manager** – graphical web interface to inspect and manage job and workflow runs, particularly useful for operations.

EVL Microservices

Microservice is a particular EVL functionality together with predefined jobs which generate jobs based only on a configuration CSV file.

- **Anonymization**
- **Data Generation**
- **Validation**
- **Key Generation**
- **Historization**
- **Staging**
- **ASN.1 decoder**
- **Write QVD**

- **Translation Microservice** – translate file formats or tables

From file/table		To file/table
CSV		CSV
JSON/XML		JSON/XML
XLS/XLSX		XLSX
QVD/QVX (Qlik)		QVD/QVX (Qlik)
Avro/Parquet		Avro/Parquet
Impala/Hive	⇒	Impala/Hive
Oracle		Oracle
Teradata		Teradata
PostgreSQL		PostgreSQL
MariaDB/MySQL		MariaDB/MySQL
mdb (MSSQL)		mdb (MSSQL)
any ODBC		any ODBC

EVL philosophy

- Component oriented with clear focus:
  "**Do One Thing and Do It Well**".
- Do not let simple things to get complex.
- Keep good balance of robustness and functionality.
- Templates and variables oriented – high level of abstraction.

EVL Anonymization Microservice

EVL Microservices are built on top of the core EVL software and retain its flexibility, robustness, high productivity, and ability to read data from various sources; including csv and Excel files, databases – Oracle, Teradata, SQL Server, etc – and Hadoop streaming data like Kafka.

EVL Anonymization Microservice provides an automated but flexible way to anonymize data.

Just based on a configuration file all anonymization jobs are generated.

History

Version	Date	Brief description
0.1	2019/03	Initial version of Anonymization Microservice. EVL version needed: = 2.0 New features: Anonymization, Unique Anonymization, Masking, Checksum, Salted checksum, Tokenization
0.2	2019/07	Focused on dates and timestamps. EVL version needed: $\geq$ 2.1 Changes: Function names accurated, dates/timestamps 9999-12-31 and 4712-12-31 are not anonymized by default New features: Encryption, Min and max can be specified also for dates and timestamps
0.3	2019/10	Next to CSV, also Excel or table can be used to store metadata. EVL version needed: $\geq$ 2.2 Changes: New features:
0.4	2020/04	
0.5	2020/10	

Table of Contents

1 Installation

1. Install EVL.
2. Login as dedicated technical user.
3. Copy microservice-anonymization.tgz into `$HOME` or any other suitable target.
4. Unpack EVL Anonymization Microservice:

```
cd           # change to $HOME or any other suitable target
tar xzf microservice-anonymization.tgz
```

5. Possibly rename folder microservice-anonymization to some more descriptive name for your project, e.g.

```
mv microservice-anonymization our_project
```

2 Jobs generation

All jobs (i.e. `evm`, `evd` and `evl` files) are completely generated based on configuration file(s)

```
anon-config.*.csv
```

by running job(s)

```
run/generate_jobs.*.evl
```

2.1 Set source/target

In fresh installation there are several `*.example` files which can be renamed (remove `.example`), edited and used. For example for one FILE source and target:

```
cd our_project
mv run/generate_jobs.file.evl.example run/generate_jobs.file.evl
```

And then simply follow the comments inside this file.

2.2 Provide information about fields

Add necessary information into related configuration file

```
anon-config.file.csv
```

For details see **anon-config** section.

2.3 Generate jobs

```
evl run/generate_jobs.file.evl
```

IMPORTANT: Every time `anon-config.file.csv` is updated, this job has to be run again!

3 Running anonymization jobs

Once jobs are generated, just run appropriate jobs, like:

```
evl run/anon.party_addr.evl
evl run/anon.party_cont.evl
evl run/anon.party_rel.evl
```

IMPORTANT: Data are always **appended** to the target!

4 anon-config

Description of each column follows:

4.1 Tenant

is not used for job generation.

4.2 Source type

is used to distinguish different inputs. Possible values are:

- `FILE` – for any supported file format, e.g. `avro`, `csv`, `json`, `parquet`, `xls`, `xlsx`, `xml`
- `ORA` – for Oracle tables
- `PG` – for PostgreSQL tables
- `TD` – for Teradata tables

When empty, then `FILE` is supposed.

4.3 Entity name

is mandatory and it is either file name or table name. It is also used for file mask of the source file(s).

4.4 Field order

is mandatory only for file sources. It specifies the order of fields, counted from 1.

4.5 Field name

is simply a field name. Case sensitive!

4.6 EVL Datatype

Field name is also used for mapping generation, so it appears in EVM mapping file.

4.7 Nullable

says if the field is nullable or not. When empty, it suppose the field nullable. Possible values are:

- `Y, y, yes, Yes, 1, True, true` – for NULLABLE
- `N, n, no, No, 0, False, false` – for NOT NULL

For file sources (e.g. csv file), when the field is NULLABLE, then an empty string is interpreted as NULL and anonymization functions keep such field again NULL.

IMPORTANT: When a field is flagged as NOT NULL, then an empty string is manipulated as proper string, so anonymization functions anonymize them. In such case from an empty string one can get non-empty anonymized value.

4.8 Minimal and maximal length

For **strings** it specifies, how long strings it should create on the output when anonymize.

By default it supposes `min_length` = 0 and `max_length` = 10, so for string fields of the length less than 10, it is actually mandatory to set max˙length. To change these defaults, one can specify other values in `project.sh` configuration file.

```
# Default Anonymization options
export EVL_ANON_DEFAULT_MIN_STRING_LENGTH=0
export EVL_ANON_DEFAULT_MAX_STRING_LENGTH=10
```

For **integer data types and decimal** it contains the range of values it should produce when doing anonymization. By default it takes `min_length` = 0 and `max_length` = 99 999 999.

4.9 Anonymization type, EVL value

These types of anonymization can be specified in the **Anonymization type** column:

- `ANONYMIZE`
- `ANONYMIZE_UNIQ`
- `TOKENIZE`

When one of these is specified and nothing else is mentioned in **EVL value** column, then it automatically generates appropriate mapping function.

When the field is of type **ANONYMIZE**, then it will use anonymize function for given data type. In general it may happen that for two different inputs it produces the same anonymized value. So it may happen that the mapping is not bijection, which is mostly the good way how to anonymize. For given value and given salt it produce the same anonymized value.

When the field is of type **ANONYMIZE_UNIQ**, then it will manage to have bijection between source field and anonymized field. So cannot happen, that one anonymized value can be produced from two different source values. For given value and given salt it produce the same token. This works **only for integer data types**, so mostly reasonable to use only for `int` and `long`. When used for other data types, it will simply use anonymization.

When the field is of type **TOKENIZE**, then at the beginning of a related job it will create so called *tokenized table*, i.e. file in `.token/` subfolder, which must have permissions 700. (Path to this folder can be changed in `project.sh` setting file.) It will then manage to have bijection between source field and anonymized field. So cannot happen, that one anonymized value can be produced from two different source values. For given value and given salt it produce the same token.

Examples:

Field name	Min len.	Max len.	Anon type	EVL value	generated function
`party_addr_line1`	10	50	`ANONYMIZE`		`anonymize(in->party_addr_line1,10,50)`
`party_addr_line1`	10	50	`ANONYMIZE`	`anonymize(IN,0,50)`	`anonymize(in->party_addr_line1,0,50)`
`party_addr_line1`	10	50		`anonymize(IN,0,50)`	`anonymize(in->party_addr_line1,0,50)`
`party_addr_line1`	10	50		`substr(IN,0,20)`	`substr(in->party_addr_line1,0,20)`

5 EVL functions

As an `EVL value` in `anon-config` file, arbitrary EVL functions and expressions can be used.

An input field is represented as `IN`.

All functions can be used in two ways:

- with pointers (preferred)
- without pointers (i.e. as referenced values, "with star")

Option with pointers is preferred as it can handle NULL values (`nullptr` in fact). So these two examples:

```
str_function(IN)
str_function(*IN)
```

are basically the same, but the first one might fail in case of using some standard C++ function and NULL value arrive. All EVL functions handle NULLs and (mostly) returns also NULL, so use `IN` for them. In all other cases use better `*IN`.

There are these two rules in all EVL string manipulation functions described in this section:

- When the first argument is a pointer, the function returns also a pointer.
- When the first argument is `nullptr`, the function returns `nullptr` as well.

5.1 Checksum functions

Standard checksum functions – `md5sum()`, `sha224sum()`, `sha256sum()`, `sha384sum()`, `sha512sum()` – can be used in mapping this way for example:

```
sha256sum(IN)
```

When the input is NULL, it returns NULL.

5.2 str_compress, str_uncompress

Compress/uncompress the given string.

Example:

```
str_compress(IN)       // snappy compression by default
str_compress(IN, compression::gzip)
str_compress(IN)       // snappy compression by default
str_compress(IN, compression::gzip)
```

When the input is NULL, it returns NULL.

5.3 str_count

Example:

```
str_count("Some text, another text.", ' ')     // returns 3
str_count("Some text, another text.", "text")  // returns 2
```

When the input is NULL, it returns NULL.

In `anon-config` it might look like this:

```
to_string(str_count(*IN, "X"))
```

5.4 str_index, str_rindex

Function returns the index (counted from 0) of the first (`str_index`) or the last (`str_rindex`) occurrence of the given substring.

When no match, then `-1` is returned.

When the string is NULL, it returns NULL.

Example:

```
str_index("Some text text", "text")   // return 5
str_index("Some text text", "xyz")    // return -1
str_index(nullptr, "x")               // return nullptr
str_rindex("Some text text", "text")  // return 10
```

5.5 str_mask_left/right

Functions return string with visible characters replaced by given character from given direction, but keep the specified number of character unchanged.

Example:

```
str_mask_left("abcd  text efgh", 6)   // returns "abcd  tex* ****"
str_mask_right("1234567890", 3, '-')  // returns "---4567890"
```

Without the second argument, asterisk (`*`) is assumed.

When the first argument is NULL, these functions return NULL.

5.6 str_pad_left/right

Add from left/right the specified character (space by default), up to the given length. It counts Bytes, not characters, so be careful with multibyte encodings.

Example:

```
str_pad_left("123",7,'0')     // returns "0000123"
str_pad_right("text",7)       // returns "text   "
str_pad_right("text",2)       // returns "text"
str_pad_left("Groß",6,'*')    // returns "*Groß" as "ß" has 2 Bytes
```

When the first argument is NULL, these functions return NULL.

5.7 str_replace

Example:

```
str_replace("Some text", ' ', '-')        // returns "Some-text"
str_replace("Some text", "Some", "Any")   // returns "Any text"
str_replace("Some text", ' ', "SPACE")    // returns "SomeSPACEtext"
```

When the first argument is NULL, it returns NULL.

In `anon-config` it might look like this:

```
str_replace(IN, ' ', '_');
```

When the input is NULL, it returns NULL.

5.8 str_to_hex, hex_to_str

Convert ordinary string to its hexadecimal representation and vice versa.

When the first argument is NULL, it returns also NULL.

Example:

```
str_to_hex("Some text")            // return "536f6d652074657874"
hex_to_str("536f6d652074657874")   // return "Some text"
```

5.9 substr

Return a substring starting after given position with the specified length.

Example:

```
substr("123456789",0,2)      // returns "12"
substr("123456789",6)        // returns "789"
```

Without the third argument, it returns the rest of the string.

When the first argument is NULL, function returns NULL.

5.10 trim, trim_left/right

Example:

```
trim("  text ")              // returns "text"
trim_left("  text ")         // returns "text "
trim_right("--text---", '-') // returns "--text"
```

Trim character `char` from both sides, from left, from right, respectively. Without the second argument, space is assumed.

In `anon-config` it might look like this:

```
trim(IN);
```

When the input is NULL, it returns NULL.

5.11 uppercase and lowercase

Example:

```
uppercase("AbCd")   // returns "ABCD"
lowercase("AbCd")   // returns "abcd"
```

When the input is NULL, it returns NULL.

Without specifying the second parameter it acts only on `A-Z` and `a-z`.

When there is a need to act also on national letters (with diacritics for example), there can be the second parameter specified with the locale:

```
uppercase(IN, "de_DE.utf8")
```

6 Custom functions

There is one example custom anonymization function in `lib/functions.cpp`:

```
anonymize_rc()
```

This particular function is country specific, it is a citizen ID in Czechia and Slovakia.

For other case specific stuff, other such can be added and used.

7 Salt

A so called **salt** is used in anonymize functions. This salt is stored in `.salt` file and must have permissions 600. (Path to this file can be changed in `project.sh` setting file.)

8 Connect to Oracle

Example of variables in `run/generate_jobs.ora.evl` which need to be set to connect to Oracle DB:

```
export SOURCE_DB="some_schema"
export TARGET_DB="some_schema"
export TARGET_TABLE_SUFFIX="_anon"

export ORAUSER="some_user"
export ORAPASS="J9_ur-76Tyx"
export ORACONN="(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
```

```
                (HOST=12.34.56.78)(PORT=1521)))(CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=abcd)))"

export TARGET_CONN="12.34.56.78:1521/abcd"
```