EVL

version 2.8

team@evltool.com

https://www.evltool.com

This manual is for **EVL** (version 2.8), a code based ETL tool.

Products, services and company names referenced in this document may be either trademarks or registered trademarks of their respective owners.

Copyright © 2017–2026 EVL Tool, s.r.o.

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts.

Table of Contents

1 Introduction

EVL originally stood for Extract–Validate–Load, but it became a fully featured ETL (Extract–Transform–Load) tool.

EVL is designed with the Unix philosophies of interoperability and "*do one thing, and do it well*" in mind.

Templates, a high level of abstraction, and the ability to dynamically create jobs, make for a powerful ETL tool.

Characteristics

- *Versatile*, i.e. cooperate with other components of the customer's solution, solving only particular problem.
- *High performance*, written in `C++`.
- *Lightweight*, just install a `rpm`/`deb` package or unzip `tgz`.
- *Highly efficient development* due to strict command-line approach.
- Managed access to the source code (e.g. `git`).
- *Linux only*, using the best of the system.
- Graphical User Interface – EVL Manager.

Features

- Natively read/write[1]:
  - File formats: CSV, JSON, XML, XLS, XLSX, Parquet, Avro, QVD/QVX, ASN.1
  - DBMS: MariaDB/MySQL, Oracle, PostgreSQL, SQLite, ODBC, (near future: Snowflake, Redshift)
  - Cloud storages: Amazon S3 and Google Storage
- Hadoop: read/write HDFS, resolve, build and run Spark jobs, Impala/Hive queries
- Partitioning, to partition data and/or parallelize processing
- Productivity boosters, to generate jobs/workflows from metadata

For the most recent information about EVL and supported formats and DBs please check `https://www.evltool.com`.

[1] Actually any source/target can be used, once available from Linux.

2 Release Notes

Versions numbering: EVL *x.y.z*

x – major release, i.e. big changes must happen to advance this number

y – minor releases, i.e. introduce new features

z – bugfixes

Overview

[Version 1.0], page 3, (2017/07)

Initial version of "classical" ETL tool with HDFS support.
New components: `Aggreg`, `Cat`, `Comp`, `Cut`, `Filter`, `Join`, `Map`, `Read`, `Sort`, `Tee`, `Trash`, `Write`.
New commands: `Mv`, `Mkdir`.

[Version 1.1], page 4, (2017/11)

Read/Write-Avro (in flat format only), Read/Write-Kafka, Read/Write-Teradata.
Validation functions within mapping: '`is_valid_*`'.
New components: `Head`, `Readavro`, `Readkafka`, `ReadTD`, `Tail`, `Validate`, `Writeavro`, `Writekafka`, `WriteTD`.

[Version 1.2], page 4, (2018/01)

Nested fields, Partitioning/parallelism, Read/Write JSON/Avro with full nested fields support.
New utilities: `evd_remove_comments`, `evd_to_avro_scheme`.
New components: `Assign`, `Depart`, `Gather`, `Merge`, `Partition`, `Readjson`, `Watcher`, `Writejson`.

[Version 1.3], page 5, (2018/03)

Read/Write XML, Records generator.
New components: `Generate`, `Readxml`, `Writexml`.
New command: `Cp`.

[Version 2.0], page 5, (2018/11)

EVL Job Manager renamed to *EVL Worklow*, man pages, Read/Write PostgreSQL, standalone components, '`in_to_out()`' enhanced, *Shared lookup* tables, Read/Write Parquet, sort within a group.
New utilities: `evl_increment_run_id` – unique EVL run ID.
New components: `Echo`, `Lookup`, `Readparq`, `ReadPG`, `RunPG`, `Sortgroup`, `Writeparq`, `WritePG`.
New commands: `Fr`, `Ls`, `Rm`, `Spark`.

[Version 2.1], page 6, (2019/04)

Microservices introduced, Read/Write Oracle, Write XLSX and QVX, Docker image available.
New components: `ReadOra`, `RunOra`, `WriteOra`, `Writeqvx`, `Writexlsx`.
New commands: `Mail`, `Test`.

[Version 2.2], page 7, (2019/10)

EVL Manager – graphical web UI, output to Qlikview/Qliksense's QVD file,
New utilities: `csv2evd`, `csv2qvd`.
New components: `Readxls`, `Readxlsx`, `Tac`, `Writeqvd`.

[Version 2.3], page 7, (2020/04)

EVL Manager enhancements, `Read` and `Write` supports also `sftp://`, `gs://`, `s3://`, *Clang* compiler can be used, *debug* mode.

New utilities: `qvd_header`.
New components: `Readqvd`.
New commands: `Chmod`.

[Version 2.4], page 7, (2020/10)
EVL Manager – Microsevices integration, crontab generation, anonymize IBAN, IP addresses manipulation, MySQL/MariaDB, password handling.
New components: `Readmysql`, `Runmysql`, `Writemysql`.
New commands: `cancel`, `crontab`, `info`, `manager`, `Sleep`.
New utilities: `json2evd`, `qvd2evd`.

[Version 2.5], page 8, (2021/04)
`Read` and `Write` supports also Samba and database URIs, randomize IBAN, EVD manipulation components.
New components: `Readevd`, `Writeevd`.

[Version 2.6], page 8, (2021/10)
New options for `Readqvd` to speed up reading subset of QVD file, secret password handling.
New mapping functions: `rsa_encrypt()`, `rsa_decrypt()`, `str_to_base64()`, `base64_to_str()`.
New commands: `key generate`, `Rmdir`.
New utilities: `pg2evd`.

[Version 2.7], page 8, (2022/04)
`Read` and `Write` supports also Google Drive: `sdrive://`.
New utilities: `evd2sql`, `guess-timestamp-format`.
New commands: `Touch`.
New components: `Readsqlite`, `Runsqlite`, `Writesqlite`.

[Version 2.8], page 9, (2023/04)
New math functions: `abs()`, `ceil()`, `floor()`, `max()`, `min()`, `pow()`, `round()`, `sqrt()`, `trunc()`
New mapping functions: `fail()`, `is_equal()`, `log()`, `warn()`
New data types: `time`, `interval`.
New command: `Calendar`.
New component: `Wc`.

Version 1.0

Released 2017/07

Changes First official version was released in the summer 2017 after more than a year of design and development and after first industry implementation in T-Mobile CZ.

New features

- ‘`Lookup tables`’ – lookup loaded into memory and used in mappings.
- ‘`Checksum functions`’ – standard checksum function for strings: ‘`md5`’, ‘`sha224`’, ‘`sha256`’, ‘`sha384`’, ‘`sha512`’.
- HDFS support

- Spark code generation – Parquet and Impala integration
- Job Manager

New components

- '`Aggreg`' – do aggregation for groups of records.
- '`Cat`' – concatenate several input flows into single output one.
- '`Comp`' – use custom component, which is actually another job.
- '`Cut`' – omit fields from input by the output data definition.
- '`Filter`' – for simple one- or two-way switch. For more complex use '`Map`'.
- '`Join`' – join two input flows by the key. Catch left/right or even unmatched records.
- '`Map`' – transform input fields and write into output fields.
- '`Read`' – read file(s) into output flow, uncompress if needed.
- '`Sort`' – sort, deduplicate, check sort; simply the output is always sorted by the key.
- '`Tee`' – replicate one input flow to several output ones.
- '`Trash`' – like `/dev/null`.
- '`Write`' – write the flow into file, compress if needed.

New commands

- '`Mkdir`'
- '`Mv`'

Version 1.1

Released 2017/11

New features

- Avro data format support (only flat structures)
- Produce/consume Kafka data stream
- Teradata FastExport and FastLoad integration
- Validation functions within mapping

New components

- '`Head`'
- '`Readavro`'
- '`Readkafka`'
- '`ReadTD`'
- '`Tail`'
- '`Validate`'
- '`Writeavro`'
- '`Writekafka`'
- '`WriteTD`'

Version 1.2

Released 2018/01

New features

- Nested fields

- Partitioning and parallelism
- Read/write JSON files with full nested fields support
- Read/write Avro files with full nested fields support

New utilities
- ‘`evd_remove_comments`’
- ‘`evd_to_avro_scheme`’

New components
- ‘`Assign`’ – assign content of the flow into variable
- ‘`Depart`’
- ‘`Gather`’
- ‘`Merge`’
- ‘`Partition`’
- ‘`Readjson`’
- ‘`Watcher`’ – makes debugging easier
- ‘`Writejson`’

Bugfixes Fixed issue with standard C++ library that was very rarely and on some systems only, causing memory/data corruption when copying partially overlapping memory regions.

Version 1.3

Released 2018/03

New features
- Read/write XML
- Sample data generator
- New string manipulation functions

New components
- ‘`Generate`’
- ‘`Readxml`’
- ‘`Writexml`’

New commands
- ‘`Cp`’

Version 2.0

Released 2018/11

New features
- EVL Job Manager renamed to EVL Workflow
- Man pages
- Standalone components (i.e. components can be used from command-line)
- Handy mapping functions applied for group of columns – ‘`in_to_out()`’
- PostgreSQL connectivity
- String manipulation functions added:
  ‘`length`’, ‘`starts_with`’, ‘`ends_with`’, ‘`substr`’,
  ‘`hex_to_str`’, ‘`str_to_hex`’,
  ‘`str_compress`’, ‘`str_uncompress`’
  ‘`str_index`’, ‘`str_rindex`’

- Shared lookup tables – one lookup can be used in several mappings
- Sort within a group – sort already grouped data
- Read/Write Parquet files
- Incremental unique Run ID

New utilities
- ‘`evl_increment_run_id`’

New components
- ‘`Echo`’
- ‘`Lookup`’
- ‘`Readparq`’
- ‘`ReadPG`’
- ‘`RunPG`’
- ‘`Sortgroup`’
- ‘`Writeparq`’
- ‘`WritePG`’

New commands
- ‘`Fr`’ – to handle File Registration
- ‘`Ls`’
- ‘`Rm`’
- ‘`Spark`’ – to run Spark jobs

Version 2.1

Released 2019/04

New features
- EVL Workflow enhanced
- Microservices – EVL parts bundled for specific purposes
- Oracle connectivity
- String manipulation functions added (‘`str_pad_left`’, ‘`str_pad_right`’)
- Docker images available
- New outputs available: XLSX, QVX

New components
- ‘`ReadOra`’
- ‘`RunOra`’
- ‘`WriteOra`’
- ‘`Writeqvx`’ – Write QlikView’s QVX file
- ‘`Writexlsx`’ – Write Excel sheets

New commands
- ‘`Mail`’ – to have e-mailing easier
- ‘`Test`’ – to handle test command also on HDFS

Version 2.2

Released 2019/10

New features
- EVL Manager – initial version of monitoring tool with web UI
- Full Unicode support (ICU)
- Helpers: ‘`csv2evd`’, ‘`csv2qvd`’

New utilities
- ‘`csv2evd`’
- ‘`csv2qvd`’

New components
- ‘`Readxls`’ – read (oldstyle) Excel sheets
- ‘`Readxlsx`’ – read Excel sheets
- ‘`Tac`’ – write records in reverse order
- ‘`Writeqvd`’ – write Qlik’s QVD file

Version 2.3

Released 2020/04

New features
- EVL Manager – various GUI enhancements
- ‘`Read`’ and ‘`Write`’ components support (next to ‘`hdfs://`’) also ‘`sftp://`’, ‘`gs://`’, and ‘`s3://`’ URI
- next to GCC, also Clang compiler can be used
- DEBUG mode

New utilities
- ‘`qvd_header`’ – get QVD’s header XML and provide various information, in JSON, XML or EVD

New components
- ‘`Readqvd`’ – read Qlik’s QVD file

New commands
- ‘`Chmod`’

Version 2.4

Released 2020/10

New features
- EVL Manager – EVL Microservices integrated
- Anonymize IBAN
- MariaDB/MySQL connectivity
- Functions: ‘`str_join`’, ‘`is_in`’
- IPv4 and IPv6 manipulation functions
- Crontab generation for scheduling workflows
- Integration of secret passwords handling

New utilities
- ‘`json2evd`’ – guess data types based on JSON file and produce EVD file

- ‘qvd2evd’ – generate EVD file based on QVD

New commands

- ‘cancel’
- ‘crontab’
- ‘info’
- ‘manager’
- ‘Sleep’ – to have sleep command integrated

New components

- ‘Readmysql’
- ‘Runmysql’
- ‘Writemysql’

Version 2.5

Released 2021/04

New features

- Read/write EVD files
- ‘Read’ and ‘Write’ components support also Samba and databases, so recognize URIs: ‘smb://’, ‘mysql://’, ‘oracle://’, ‘postgres://’, ‘teradata://’

New components

- ‘Readevd’
- ‘Writeevd’

Version 2.6

Released 2021/10

New features

- RSA encryption/decryption by mapping function: `rsa_encrypt()`, `rsa_decrypt()`, `str_to_base64()`, `base64_to_str()`
- New utility `pg2evd` to create EVL data structure (EVD file) based on PostgreSQL table.

Version 2.7

Released 2022/04

New features

- Read/Write SQLite table and run SQL in SQLite DB
- ‘Read’ and ‘Write’ components support also Google Drive, available by URI schema ‘gdrive://’ and SQLite DB, with URI schema ‘sqlite://’

New components

- ‘Readsqlite’
- ‘Runsqlite’
- ‘Writesqlite’

New utilities

- ‘evd2sql’ – produce CREATE TABLE statement based on EVD
- ‘guess-timestamp-format’ – return C-style timestamp format string based on the input timestamps

Version 2.8

Released 2023/04

New mapping functions

- ‘`fail()`’, ‘`is_equal()`’, ‘`log()`’, ‘`warn()`’
- Math functions: ‘`abs()`’, ‘`ceil()`’, ‘`floor()`’, ‘`max()`’, ‘`min()`’, ‘`pow()`’, ‘`round()`’, ‘`sqrt()`’, ‘`trunc()`’

New command

- ‘`Calendar`’ – To specify a calendar based on which the job or workflow will be fired

New component

- ‘`Wc`’ – Count records and bytes

New data types

- ‘`interval`’
- ‘`time`’

3 Installation and Settings

EVL Installation

- Section 3.1 [Linux RPM], page 10, – installation on RedHat-like systems,
- Section 3.2 [Linux DEB], page 10, – installation on Debian-like systems,
- Section 3.3 [Other Unix Systems], page 10, – installation on MacOS, etc.

Settings after installation

- Section 3.4 [Settings], page 11, – to set various environment variables,
- Section 3.5 [Text Editor], page 12, – syntax highlighting in various editors.

3.1 Linux – RPM

i.e. RedHat, CentOS, Fedora, Oracle Linux.

For *CentOS 8* firstly install required packages from *powertools* repo:

```
sudo dnf -y install dnf-plugins-core
sudo dnf config-manager --set-enabled powertools
sudo dnf install --enablerepo=powertools snappy-devel
```

To install all the libraries:

```
sudo dnf install -y gcc gcc-c++ zlib-devel libxml2 snappy-devel libicu \
  gettext findutils binutils coreutils procps-ng \
  info man sqlite bash-completion libaio openssl \
  boost-iostreams boost-filesystem boost-regex \
  boost-system boost-program-options
```

Use `yum` instead of `dnf` on older systems.

Then install EVL package itself. Package name might have different name, it depends on the system and/or edition. For example for CentOS version 7 it would be:

```
sudo dnf install evl-2.8.1-1.el7.x86_64.rpm
```

3.2 Linux – DEB

i.e. Ubuntu, Debian, etc.

```
sudo apt-get install gcc g++ zlib1g-dev libxml2 libsnappy-dev libicu-dev \
  gettext-base binutils coreutils bsdmainutils procps \
  info man-db sqlite bash-completion |
  dos2unix libncurses-dev libncurses5-dev
```

Then install EVL package itself. Package name might have different name, it depends on the edition. For example for Debian it would be:

```
sudo apt-get install evl-utils_2.8.1-1_all.deb
sudo apt-get install evl-tool_2.8.1-1_amd64.deb
```

3.3 Other Unix systems

i.e. Mac OS, etc.

Basically any standard Unix system with Bash and couple of standard utilities (gettext, binutils, coreutils) and libraries (boost, snappy, xml2, etc.) is possible.

Ask `support@evltool.com` for help.

3.4 Settings

EVL installation resides (usually) in

```
/opt/evl
```

To initiate EVL for current user, run

```
/opt/evl/bin/evl --init
```

which adds an `.evlrc` file into your `$HOME` folder and adds sourcing it into `$HOME/.bashrc`.

Then one can check, add or modify some settings in `.evlrc`, for example variable `EVL_ENV`. These settings are top level settings for given user. (Later there are `project.sh` files in each project, to set project-wide variables.)

After that EVL is ready to use. Good to start is to create new project with some sample data, jobs and workflows:

```
evl project sample my_first_sample
```

3.4.1 Compiler

Mappings are compiled either by GCC or Clang. It depends on environment variable `EVL_COMPILER`, these two values are possible:

```
EVL_COMPILER=gcc
EVL_COMPILER=clang
```

If this variable is not set, then on Linux systems is GCC by default, and on Windows and Mac it is Clang.

GCC must be at least in the version 7.4 and Clang at least 6.0.

When Clang would be the option, one can replace gcc/c++ packages by clang above in installation instructions.

3.4.2 Project

EVL project is a directory with EVL jobs, workflow, data structure definitions, mappings, etc. Each project is intended for a group of jobs and workflows which are grouped somehow from the business point of view. So completely unrelated processings, which share nothing, would be good to place in separate projects.

project.sh

Each project has `project.sh` file inside. This file contains project-wide settings. If the project is created by `evl project new` or by `evl project sample` command, then `project.sh` would contain a good set of variables to start with.

Path to `log` and `tmp` folders are handled by environment variables:

`EVL_PROJECT_LOG_DIR`
: path to folder where log files are stored,

`EVL_PROJECT_TMP_DIR`
: path to folder where temporary files are stored.

By default are these variables set to:

```
EVL_PROJECT_LOG_DIR="$EVL_LOG_PATH/<project_name>"
EVL_PROJECT_TMP_DIR="$EVL_TMP_PATH/<project_name>"
```

These values can be override in `project.sh` file.

Project directory structure

In each project directory, there would be these files and folders:

`build/`
: contains subdirectories of compiled components of each job (one subfolder equals to one `evl` file).

`doc/`
: generated and/or custom documentation of the project.

`evc/*.evc`
: EVL custom component definition files.

`evd/*.evd`
: EVL data definition files.

`evs/*.evs`
: EVL job structure files (job definition itself, sometimes called a graph of ETL process).

`evm/*.evm`
: EVL mapping files for components `Aggreg`, `Join` and `Map`.

`ews/*.ews`
: EVL workflow structure files.

`job/*.evl`
: files, called by `evl run` command, which specify the variables for the job structure, i.e. it contains parameters of the job described by `evs` file.

`workflow/*.ewf`
: EVL workflow parameter files for `ews` files.

`project.sh`
: This file is interpreted as `bash` shell script at the beginning of each job in the project. Usually contains all project wide variables.

3.5 Text Editor

All you need to work with EVL is having a text editor.

Whatever text editor is your favourite, it is good to set syntax highlighting this way:

Syntax	**File mask**
Bash	`*.evs`, `*.evl`, `*.evc`, `*.ewf`, `*.ews`
C++	`*.evm`, `*.evd`

3.5.1 Vim

To achieve syntax to be properly highlighted in Vim, just add these lines into your `~/.vimrc` file:

```
" EVL settings "
autocmd BufRead,BufNewFile *.ev[slc] set syntax=sh
autocmd BufRead,BufNewFile *.ew[fs] set syntax=sh
autocmd BufRead,BufNewFile *.ev[md] set syntax=c
```

4 EVL Overview

4.1 ETL in general

ETL stands for Extract–Transform–Load and shortly said it is a system to move and transform data between data storages.

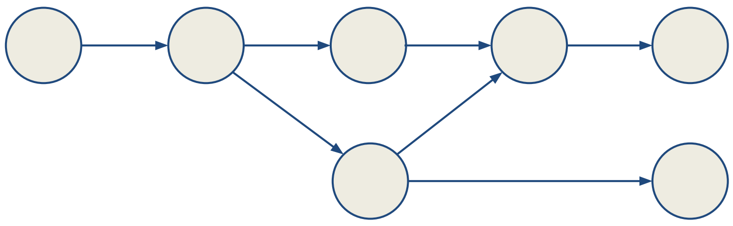
ETL processing usually consists of three main parts

- **ETL** itself (ETL jobs) – to process data
- **Orchestration** (ETL workflows) – to manage ETL jobs, handle job consequences, await file delivery, provide information about processing via e-mail or SNMP traps, etc.
- **Scheduling** – to fire ETL workflows at give time in a given day

Quite often is *Orchestration* and *Scheduling* named together as *Scheduler*, but let's distinguish these two parts of ETL system to follow Unix Philosophy: "*do one thing, and do it well*".

ETL jobs or ETL workflows consists of one or more oriented acyclic graphs, named more often as **DAG = Directed Acyclic Graph**.

So the following DAG may describe either ETL job or ETL workflow.



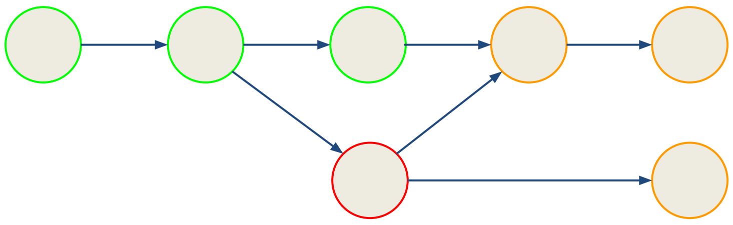
The only difference is the meaning of vertices and edges:

	jobs	**workflows**
vertices	data modifying components	jobs, other workflows
edges	data flows	successor

So then the approach in restarting of jobs and workflows is also different:

- When ETL job fails, whole must be restarted.
- When ETL workflow fails, can be either restarted from the beginning or continue from last failure(s).

So an ETL workflow like this:

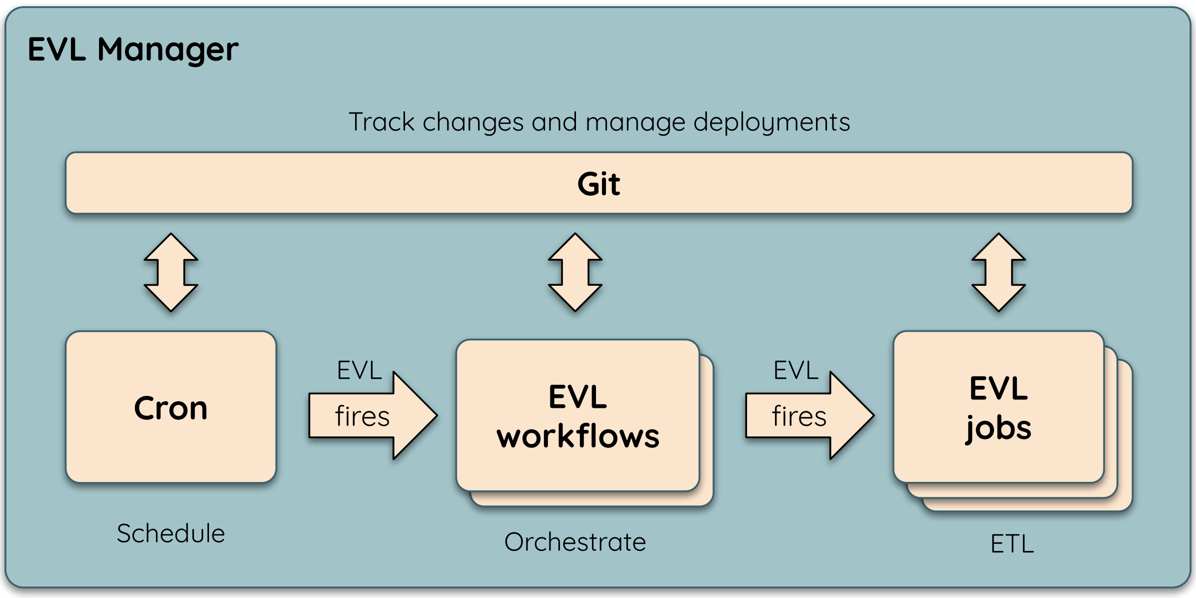


might be restarted from the red job. Green (i.e. successful ones) will be skipped.

4.2 EVL approach

Considering above theory, EVL splits ETL system into three main entities:

- Section 4.4 [EVL Jobs], page 15,
- Section 4.5 [EVL Workflows], page 15,
- Section 4.6 [Scheduling], page 15,



All three entities are supposed to be tracked by **Git** or any other version control system.

4.3 Terminology

Project Scope of work isolated from the business perspective. EVL supports inter-project workflow dependencies, but often it delivers data from the source to the final target independently. Technically is the project represented by one directory and the name of the directory is the name of the project. In this directory is `project.sh` (bash) file, `crontab.sh` and directories of all other EVL configuration files:

`evc/`	EVL custom Components, a common piece of a job
`evd/`	EVL Data Definitions, data types, separators, null flags, encoding, etc., simply something like DDL
`evm/`	EVL Mappings, C++ syntax
`evs/`	EVL Structure file, i.e. templates of jobs
`ews/`	EVL Workflow Structure, i.e. templates of workflows
`job/`	`*.evl` files contain only parameters for given `*.evs` file, these files are called by `evl run` command. There might be also standard bash scripts `*.sh`
`workflow/`	`*.ewf` files contain only parameters for given `*.ews` file, to be called by command `evl run`

Important: Files `*.evc`, `*.evs`, `*.ews`, `*.evl` and `*.ewf` are interpreted as bash scripts.

Task Either job or workflow or wait for a file. Each run of the task has its so-called **Order Date** (ODATE) and **Run ID** (unique increment integer per project).

Job The basic unit of work, it can be either a standard Bash script `*.sh` or it can be represented by an `*.evl` parameters file which uses one and only one EVS job template. The job is always restarted from the beginning, (despite in some cases, like downloading X files from the source, it can start after the last successful step, it depends on how the job is written). So jobs are physically represented by `*.evl` and `*.sh` files.

Workflow List of task calls with dependencies split by waits for another workflows (even from other projects). A project can have one or multiple workflows physically represented by `*.ewf` files. Workflow can be restarted from the beginning or from the last successful step.

Scheduling
Workflows, and jobs (if not included in workflows) can be scheduled using crontab (or any other scheduler). For crontab there is `crontab.sh` script to wrap scheduling up.

4.4 EVL Jobs

4.5 EVL Workflows

4.6 Scheduling

5 Main EVL Command

All command line EVL functionality is handled by the main command '`evl`', which either run EVL jobs and workflows or serves other EVL subcommands.

`evl project`
: handles EVL projects, like create new or sample one, source variables from '`project.sh`', or get particular project variables. For details, check '`man evl-project`'.

`evl <evl_command>`
: it calls particular EVL command/component, like '`sort`', or '`readjson`'. All possible EVL components or commands are listed below and each has its own man page which explain usage and arguments. To see man page for a command, run '`man evl-<evl_command>`'.

`evl run ( <job>.evl | <workflow>.ewf | <script>.sh )`
: To run an EVL job or workflow, for details check '`man evl-run`'. as the most common usage is to run a job, there is a shortcut:

```
evl job/<job>.evl
```

5.1 Usage

```
evl
  <evl_command>  [<option>...]

evl
  <evl_command>  ( --help | --usage )

evl
  ( --init | --expiration-date | --help | --usage | --version )
```

5.2 Examples

1. To run a job with yesterday Order Date:

```
evl job/staging.invoices.evl --odate=yesterday
```

2. To run a workflow with yesterday Order Date:

```
evl run workflow/staging.ewf --odate=yesterday
```

5.3 Options

`--init`
: to initiate EVL installation under your (i.e. non-root) user. To be run only once for each user, it creates or overwrite '`$HOME/.evlrc`' file.

`--expiration-date`
: return an expiration date of this version of EVL, empty output means no expiration

Commands for base and mapping components:

`aggreg`
: aggregate (and map) records by key

`assign`
: assign the content of input flow or file into specified variable

`cat`	concatenate flows or files
`cmd`	run any system command with possibility to connect to flow
`comp`	run custom EVL component
`cut`	remove columns from input flow or file
`departition`	gather or merge partitioned flows or files into one partition
`echo`	write an argument into output flow or file
`filter`	split flows according to a condition or just filter records out
`gather`	gather multiple flows or files into one in round-robin fashion
`generate`	create artificial records
`head`	output the first part of input flow or file
`join`	join sorted inputs
`lookup`	create and remove shared lookup
`map`	generic mapping
`merge`	merge sorted inputs (by keeping the sort)
`partition`	partition input flow or file
`sort`	sort (and possibly deduplicate) records of input flow or file
`sortgroup`	sort input flow or file within a group
`tac`	write flow or file in reverse
`tail`	output the last part of input flow or file
`tee`	replicate input flow or file

`trash`
: send flow(s) to /dev/null

`uniq`
: deduplicate sorted input flow or file

`validate`
: check data types and possibly filter out invalid records

`watcher`
: catch flow content into text file, debugging purpose

Commands for read components:

`read`
: generic source reader, handle various file types (‘`Avro`’, ‘`CSV`’, ‘`json`’, ‘`Parquet`’, ‘`QVD`’, ‘`xls`’, ‘`xlsx`’ and ‘`xml`’), compression (‘`gz`’, ‘`tar`’, ‘`bz2`’, ‘`zip`’, ‘`Z`’) and URI Scheme for file storage (‘`file:`’, ‘`gdrive:`’, ‘`gs:`’, ‘`hdfs:`’, ‘`s3:`’, ‘`sftp:`’, ‘`smb:`’) and for tables (‘`mysql:`’, ‘`mssql`’, ‘`postgres:`’, ‘`oracle:`’, ‘`sqlite:`’, ‘`teradata:`’)

`readasn1`
: read ASN.1 format

`readavro`
: read and parse Avro file format

`readevd`
: read and parse EVD file

`readjson`
: parse JSON input

`readkafka`
: consume Kafka topic

`readmssql`
: read MS SQL table into flow or file

`readmysql`
: read MariaDB/MySQL table into flow or file

`readora`
: read Oracle table into flow or file

`raedparquet`
: read Parquet files

`readpg`
: read PostgreSQL table into flow or file

`readqvd`
: read and parse QVD (QlikView, Qlik Sense) file

`readredshift`
: read Amazon Redshift table into flow or file

`readsqlite`
: read SQLite table into flow or file

`readtd`
: read Teradata table into flow or file

`readxls`
: read XLS (MS Excel) sheet

`readxlsx`
: read XLSX (MS Excel) sheet

`readxml`
: parse XML input

Commands for run SQL components:

`runImpala`
: run impala sql from file or from input

`runmssql`
: run SQL in MS SQL database

`runmysql`
: run SQL (or mysql command) in MariaDB/MySQL database

`runpg`
: run SQL in Oracle

`runpg`
: run SQL or psql command in PostgreSQL database

`runredshift`
: run SQL query in Amazon Redshuft database

`runsqlite`
: run SQL (or sqlite3 command) in SQLite database

Commands for write components:

`write`
: generic file and table writer, handle various file types (‘`Avro`’, ‘`CSV`’, ‘`json`’, ‘`Parquet`’, ‘`QVD`’, ‘`QVX`’, ‘`xlsx`’ and ‘`xml`’), compression (‘`gz`’, ‘`bz2`’, ‘`zip`’) and URI Scheme for file storage (‘`file:`’, ‘`gdrive:`’, ‘`gs:`’, ‘`hdfs:`’, ‘`s3:`’, ‘`sftp:`’, ‘`smb:`’) and for tables (‘`mysql:`’, ‘`mssql`’, ‘`postgres:`’, ‘`oracle:`’, ‘`sqlite:`’, ‘`teradata:`’)

`writeavro`
: write input as Avro file

`writeevd`
: write EVD file in proper format

`writejson`
: write input as JSON

`writekafka`
: produce Kafka topic

`writemssql`
: write flow or file into MS SQL table

`writemysql`
: write flow or file into MariaDB/MySQL table

`writeora`
: write flow or file into Oracle table

`writeparquet`
: write flow or file into Parquet files

`writepg`
: write flow or file into PostgreSQL table

`writeqvd`
: write flow or file into QVD (QlikView, Qlik Sense) file

`writeqvx`
: write flow or file into QVX (QlikView, Qlik Sense) file

`writeredshift`
: write flow or file into PostgreSQL table

`writesqlite`
: write flow or file into SQLite table

`writetd`
: write flow or file into Teradata table

`writexlsx`
: write flow or file into XLSX (MS Excel) files

`writexml`
: write input as XML

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

5.4 Environment

The list of all EVL variables with their default values. One can change these values in his '`~/.evlrc`' file or in the project in '`project.sh`'.

`EVL_BUILD_COMP=1`
: whether to build the job every time it runs or not. In production it is mostly safe to set to '`0`', so the job is then built only the first time, and then only if the source files changed.

`EVL_COLOURS=1`
: terminal output use colours, but in the case that it cause troubles, one can switch it off by setting environment variable '`EVL_COLOURS=0`'

`EVL_COMPILER=gcc`
: mappings are compiled either by GCC or Clang. By this variable one can specify which one to use. Possible valus are:

```
EVL_COMPILER=gcc
```

```
EVL_COMPILER=clang
```

If this variable is not set, then on Linux systems is GCC used by default, and on Windows and Mac it is Clang.

GCC must be at least in the version 7.4 and Clang at least 6.0.

`EVL_COMPILER_PATH`
: path to GCC's or Clang's '`bin`', '`include`', '`lib`' and '`lib64`' folder. Leave empty to use system-wide GCC/Clang.

`EVL_CONFIG_FIELD_SEPARATOR=';'`
: the default field separator used in config files when no '`sep=`' attribute for a field in EVD file, use this character instead. This character might be any one of the first 128 ascii ones.

`EVL_DEBUG_FAIL_RECORD_NUMBER=2`
: the number of records to show when fail with '`EVL_DEBUG_MODE=1`'

`EVL_DEBUG_MODE=0`
: if set to 1, then it checks if you try to assign NULL value into not nullable field, and provide the most recently processed records in case of a failure. But it slows down the processing, so use only in developmnet or switch on temporarily in production in the case of investigation data problems.

`EVL_DEFAULT_FIELD_SEPARATOR='|'`
: when no '`sep=`' attribute for a field in EVD file, use this character instead. This character might be any one of the first 128 ascii ones.

`EVL_DEFAULT_RECORD_SEPARATOR`
: when no '`sep=`' attribute for the last field in EVD file, use this character instead. This character might be any one of the first 128 ascii ones. By default a Linux newline is used:

```
EVL_DEFAULT_RECORD_SEPARATOR=$'\n'
```

but to use Windows end of line (i.e. '`\r\n`'), use components' options '`--text-input-dos-eol`' and/or '`--text-output-dos-eol`'.

`EVL_DEFAULT_DECIMAL_SEPARATOR="."`
: decimal places separator for decimal data type, by default it is a dot. E.g. textual representation of decimal field would look like '`21872.88`', and with decimal separator '`,`' would look like '`21872,88`'.

`EVL_DEFAULT_THOUSANDS_SEPARATOR=""`
: for decimal data type thousands separator can be used, turned off by default. E.g. textual representation of decimal field would look like '`21872.88`'. Setting thousands separator to '`,`' produce textual representation '`21,872.88`'.

`EVL_DEFAULT_DATE_PATTERN="%Y-%m-%d"`
: default date format string, e.g. '`2024-11-07`'

`EVL_DEFAULT_DATETIME_PATTERN="%Y-%m-%d %H:%M:%S"`
: default datetime format string, e.g. '`2024-11-07 07:12:29`'

`EVL_DEFAULT_TIMESTAMP_PATTERN="%Y-%m-%d %H:%M:%E*S"`
: default timestamp format string, e.g. '`2024-11-07 07:12:29.123456789`'

`EVL_DEFAULT_TIME_PATTERN="%H:%M:%S"`
: default time format string, e.g. '`07:12:29`'

`EVL_DEFAULT_TIME_NANO_PATTERN="%H:%M:%E*S"`
: default time_ns format string, e.g. '`07:12:29.123456789`'

`EVL_ENV=DEV`
: to specify an environment, usually one of '`DEV`', '`TEST`' or '`PROD`'.

`EVL_FASTEXPORT_SLEEP, EVL_FASTEXPORT_TENACITY, EVL_FASTEXPORT_SESSIONS`
: Teradata FastExport options.

`EVL_FASTLOAD_ERROR_LIMIT, EVL_FASTLOAD_SESSIONS`
: Teradata FastLoad options.

`EVL_FR=1`
: if set to 0, then EVL File Register is not used, only provide debug messages, but does nothing.

`EVL_FR_LOG_FILE`
: file to be used for storing information for EVL File Register.

`EVL_KAFKA_CONSUMER_COMMAND, EVL_KAFKA_PRODUCER_COMMAND`
: paths to Kafka consumer and producer commands.

`EVL_LOG_DIR="$HOME/evl-log"`
: path to logs from job and workflow runs. The default is set in '`/opt/evl/etc/evlrc`'.

`EVL_MAIL_SEND=1`
: send e-mails by default in the case of fails in a workflows or by the commmand Mail. To switch off, for example in non-production environments, set '`EVL_MAIL_SEND=0`'.

`EVL_MONITOR_DBMS="sqlite"`
: by default SQLite DB is used, but for production environment PostgreSQL recommended. In such case use '`postgres`' value for this variable.

`EVL_MONITOR_ENABLED=1`
: monitoring logging can be turned off by setting this variable to 0.

`EVL_MONITOR_POSTGRES_DB="evl_monitor", EVL_MONITOR_POSTGRES_HOST="localhost",`
EVL_MONITOR_POSTGRES_PORT=5432, EVL_MONITOR_POSTGRES_USER="evl_monito
: connection information when PostgeSQL DB is used for logging monitoring entries.

`EVL_MONITOR_SQLITE_TIMEOUT=2000`
: when SQLite DB is used for logging monitoring entries, this value is used for timeout for SQLite.

`EVL_MONITOR_SQLITE_PATH="$EVL_LOG_DIR"`
: path to SQLite database for EVL Manager. The default is set in '`/opt/evl/etc/evlrc`'.

`EVL_NICE=1`
: each EVL command and component is fired prefixed by:

```
eval nice -n $EVL_NICE
```

To change the priority of EVL processes, to have EVL jobs "nicer", one can set '`EVL_NICE`' to the value between 0 and 19. Higher number means that processes will have lower priority. For details one can check '`man nice`'.

`EVL_ODATE`
: when no '`--odate=`' option is used when running a job or workflow, it tries to use an Order Date from this variable. So calling:

```
evl job/some_job.evl --odate=20260121
```

is the same as:

```
export EVL_ODATE=20260121
evl job/some_job.evl
```

`EVL_PARTITIONS`
: to specify how many partitions to use in ‘`Partition`’ component. This EVL installation allows at most ‘`1024`’ partitions.

`EVL_PASSFILE="$HOME/.evlpass"`
: contains path to file with passwords. Must have ‘`600`’ permissions. Structure of the file:

```
server:port:database:username:encrypted_password
```

So for example:

```
10.0.0.10:5432:some_db:some_user:ka786_Ufzf5oaD9
10.0.0.10:1521:some_db:some_user:ka786_Ufzf5oaD9
100.10.9.8:22:/target/folder:user:LKKo-098
localhost:3001:impala_user:2_lLkPl_010
212.0.0.11:288:USR_0000:162534
```

For details see ‘`man evl-password`’.

`EVL_PROCESSES_CHECK_SEC=0.4`
: how often (in seconds) check processes if they are still running. For very long running jobs it makes sense to increase this value to even 2.0 seconds. This default value is good for jobs with many steps (i.e. many Wait components) and quite short processing so each step finish as soon as possible. Possible range is from 0.1 to 2.0.

`EVL_PROGRESS_REFRESH_SEC=2`
: when ‘`--progress`’ option is used, it refresh the state every 2 seconds by default. To change this default, set this variable to other number of seconds. Possible range is from 1 to 30.

`EVL_PROJECT_LOG_DIR`
: by default project’s log directory is set to:

```
EVL_PROJECT_LOG_DIR="$EVL_LOG_DIR/<project_name>"
```

`EVL_PROJECT_TMP_DIR`
: by default project’s temporary directory is set to:

```
EVL_PROJECT_TMP_DIR="$EVL_TMP_DIR/<project_name>"
```

`EVL_RUN_ID_FILE`
: path to file which stores incremental ‘`RUN_ID`’, a unique ID of each job or workflow run. It is unique within a project. By default it is:

```
EVL_RUN_ID_FILE="$EVL_PROJECT_LOG_DIR/evl_run_id.hwm"
```

`EVL_TMP_DIR="/tmp"`
: path to (local) temporary directory, to be used by jobs and workflows. Situate this folder on the same mount point as data will be, to make ‘`mv`’ command fastest as possible. The default is set in ‘`/opt/evl/etc/evlrc`’.

`EVL_TRACE_LEVEL=0`
: specify number between 0 and 3 to say how detailed EVL Trace Messages should be:

```
0 - do not display trace messages
1 - code to be copy+paste and run from command line
2 - what is going to be enter to monitoring table or log
3 - very detailed information about PIDs numbers etc.
```

`EVL_WATCHER=0`
: whether or not the component ‘`Watcher`’ is silent. In production this would be usually set to ‘`0`’, but in development, if ‘`Watcher`’ is used to investigate interim data, it is fine to set to ‘`1`’. Check ‘`man evl-watcher`’ for more details.

5.5 evl project

Create new EVL project(s) or get project settings. Consider current directory as a project one, unless `<project_dir>` is specified with either full or relative path. Last folder in the `<project_dir>` path is considered as project name. Prefer to use small letters for project names, however numbers, capital letters, underscores and dashes are possible.

Projects can be included into another projects. But remember that parent's project.sh is not automatically included (i.e. sourced) by subproject's one.

`create <project_name> [<project_name_2>...]`
: create <project_name> directory (directories) with standard subfolders structure and default '`project.sh`' configuration file.

`create --sample <project_name> [<project_name_2>...]`
: create <project_name> directory (directories) with sample data and sample jobs and workflows.

`get <variable_name> [--path] [--omit-newline] [--project=<project_dir>]`
: get the value of <variable_name>, based on the project.sh configuration file. Search '`project.sh`' in the current directory, unless <project_dir> if mentioned. With option '`--path`', it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning). With option '`--omit-newline`', return value without trailing newline.

To drop the whole project simply delete the folder recursively.

Synopsis

```
evl project create
  <project_name>... [--sample]
  [-v|--verbose]

evl project get
  <variable_name>
  [-p|--project=<project_dir>]
  [--path] [--omit-newline]
  [-v|--verbose]

evl project
  ( --help | --usage | --version )
```

Options

`--omit-newline`
: return value without trailing newline, good for example for assigning returned value into a variable

`--path`
: it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning)

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--sample`
: create project with sample configuration

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To create three main projects with couple of subprojects:

```
# shared to all projects
evl project create shared

evl project create stage       # shared stuff only for "stage" projects
evl project create stage/sap stage/tap stage/erp stage/signaling

evl project create dwh         # shared stuff only for "dwh" projects
evl project create dwh/usage dwh/billing dwh/party dwh/contract dwh/product

evl project create mart        # shared stuff only for "mart" projects
evl project create mart/marketing mart/sales
```

2. To create new project with sample data, jobs and workflows:

```
evl project create --sample my_sample
```

3. To get the project path to log directory (i.e. ‘`EVL_PROJECT_LOG_DIR`’):

```
evl project get --path EVL_PROJECT_LOG_DIR
```

5.6 evl run

See Section 13.16 [Run], page 157, for details.

5.7 evl workflow

EVL Workflow is a code based orchestration tool. It fires EVL tasks in parallel and in specified order on specified target host and consider specific priorities.

EVL task

Task is one of the following:

`Shell Script (‘*.sh’)`
: any shell script with ‘`.sh`’ suffix

`EVL job or workflow (‘*.evl’ or ‘*.ewf’)`
: EVL job is an ETL job, i.e. one or more DAGs (Directed Acyclic Graph) with data flows on edges and data modifying components as vertices. EVL workflow is also one or more DAGs, but vertices are Tasks (i.e. Shell Scripts, EVL jobs, other EVL workflows, or Wait for a file), edges are successors.

`Wait for a file`
: to sniff for an existence of a file with given file mask.

The workflow consists (mostly) of '`Run`' components, which are used in EWS workflow structure definition file, and which fires EVL jobs or other EVL workflows or wait for a file with given file mask. For details about this component, see '`man evl-run`'.

'`EWS`' is EVL workflow structure file (workflow template), for details see '`man 5 evl-ews`'.

'`EWF`' is EVL worflow definition file (a workflow), for details see '`man 5 evl-ewf`'.

Arguments

`run`
: run `<workflow>` with Order Date ('`ODATE`') equal to `<odate>`. In case that workflow with given '`ODATE`' has been started in the past, it will fail. Use '`continue`' or '`restart`' in such cases. This command is intended to be scheduled by '`crontab`' for example.

`continue`
: continue `<workflow>` with Order Date equal `<odate>` from last failed step, i.e. do not run again already successfully finished steps. This command is useful for usual manual restart from failed point.

`restart`
: restart whole `<workflow>` (with given ODATE) from the beginning, no matter what is the status of the workflow. Use this command with care, normally not to be used in production environment.

Order Date

is a date for which the data are being processed. Every workflow has to be run with some `<odate>`. When no `<odate>` is specified, then current date is used. An `<odate>` can be of any form that standard GNU/Linux command '`date`' can recognize as a date. Recommended is however to use format '`YYYYMMDD`' or '`yesterday`'.

Synopsis

```
evl
  ( run | continue | restart ) <workflow>...
  [-D|--define=<variable=value>]...
  [-o|--odate=<odate>] [-p|--project=<project_dir>]
  [-s|--progress] [-v|--verbose]

evl workflow
  ( --help | --usage | --version )
```

Options

`-D, --define=<definition>`
: the `<definition>` is evaluated right before running a workflow, but after evaluating settings from '`ewf`' file, e.g. '`-DSOME_PATH=/some/path`' will do '`eval SOME_PATH=/some/path`', and overwrites then variable SOME_PATH possibly defined in '`ewf`' file. Multiple '`--define`' options can be used.

`-o, --odate=<odate>`
: run evl workflow with specified `<odate>`, environment variable '`EVL_ODATE`' is ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`-s, --progress`
: it shows the states of each component, refreshed every ‘`$EVL_PROGRESS_REFRESH_SEC`’ seconds. (2 seconds by default.)

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Commands

EVL workflow structure file (‘`*.ews`’ file) is resolved as Bash script. Following EVL commands can be used, see ‘`man evl-<command>`’ for details.

`calendar`
: continue based on specified calendar file

`Chmod`
: change file permissions, act by URI (`file://`, hdfs://, `sftp://`)

`Cp`
: copy files, act by URI (`file://`, gdrive://, gs://, hdfs://, s3://, `sftp://`, `smb://`)

`end`
: end up an EVL job or workflow structures (‘`EVS`’ or ‘`EWS`’ files)

`Ls`
: list directory contents, act by URI (`file://`, gdrive://, gs://, hdfs://, s3://, `sftp://`, `smb://`)

`Mail`
: send an e-mail

`Mkdir`
: create directory, act by URI (`file://`, hdfs://, s3://, `sftp://`)

`Mv`
: move (rename) files, act by URI (`file://`, gdrive://, gs://, hdfs://, s3://, `sftp://` , `smb://`)

`Rm`
: remove files or directories, act by URI (`file://`, gdrive://, gs://, hdfs://, s3://, `sftp://`, `smb://`)

`Rmdir`
: remove empty directories, act by URI (`file://`, hdfs://, `sftp://`)

`Sleep`
: run previously defined EVL tasks and delay for a specified amount of time

`Snmp`
: send a SNMP trap message

`Test`
: check file types and existence, handle also hdfs and AWS S3.

`Touch`
: change file timestamp, create file if not exist, act by URI (`file://`, hdfs://, `sftp://`)

`Wait`
: split pieces of EVL job or workflow into steps

Run Component

EVL workflow structure file (EWS file) is resolved as Bash script. Next to *Commands* above, which are run immediately, there is a '`Run`' component which is just parsed, but fired later once '`Wait`' or '`End`' command is reached.

For details see '`man evl-run`'.

Environment Variables

The list of variables which controls EVL workflow behaviour. With their default values. These variables can be set for example in user's '`~/.evlrc`' file or in the project's '`project.sh`'.

`EVL_RUN_FAIL=1`
: whether or not to fail given '`Run`' command once an EVL task fails, so when zero is set, the '`Run`' command continue regardless task failures

`EVL_RUN_FAIL_MAIL=1`
: whether or not to send an e-mail when the task fails

`EVL_RUN_FAIL_MAIL_SUBJECT="Project '$EVL_PROJECT' FAILED"`
: subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_FAIL_MAIL_MESSAGE`
: message of such e-mail, by default it is:

```
Project:      $EVL_PROJECT
Top Level WF: $EVL_WORKFLOW_TOP
Current WF:   $EVL_WORKFLOW
Task:         $EVL_TASK
Order Date:   $EVL_ODATE
Sent to:      $EVL_MAIL_TO
Task log:     $EVL_TASK_LOG
Tail of log:  $(tail $EVL_TASK_LOG)
```

where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_FAIL_SNMP=0`
: whether or not to send SNMP trap when the task fails.

`EVL_RUN_FAIL_SNMP_MESSAGE='$EVL_PROJECT FAILED'`
: SNMP message to be send.

`EVL_RUN_RETRY=0`
: the number of times it retries to run the task again. Zero means no retry and fail '`Run`' command once the given task fails.

`EVL_RUN_RETRY_INTERVAL=5m`
: the amount of time between retries. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_TARGET_TYPE=local`
: where to run EVL task(s), possible values are '`k8s`', '`local`', and '`ssh`'.

`EVL_RUN_TIME=24h`
: maximal run time, so if the task invoked by '`Run`' command is not finished after this amount of time, it is killed. The time is counted since the task is really running, not since the invocation (i.e. waiting time is not included). It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK=1`
: whether or not to wait for a lock file, i.e. if somebody is running the same task at the moment.

`EVL_RUN_WAIT_FOR_LOCK_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK_TIME=10h`
: maximal amount of time to wait for a lock file. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE=0`
: whether or not to automatically wait for previous Order Date of given task. Setting to 1 might be useful when you must run daily processing strictly in right order.

`EVL_RUN_WAIT_FOR_PREV_ODATE_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE_TIME=10h`
: maximal amount of time to wait for previous Order Date. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WARN_MAIL=0`
: whether or not to send an e-mail when there is warning

`EVL_RUN_WARN_MAIL_SUBJECT='$EVL_PROJECT WARNING'`
: subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_WARN_MAIL_MESSAGE`
: message of such e-mail, by default it is:

```
Project:       $EVL_PROJECT
Top Level WF:  $EVL_WORKFLOW_TOP
Current WF:    $EVL_WORKFLOW
Task:          $EVL_TASK
Order Date:    $EVL_ODATE
Sent to:       $EVL_MAIL_TO
```

```
Task log:     $EVL_TASK_LOG
Tail of log:  $(tail $EVL_TASK_LOG)
```

where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_WARN_SNMP=0`
: whether or not to send SNMP trap when there is a warning

`EVL_RUN_WARN_SNMP_MESSAGE='$EVL_PROJECT WARNING'`
: SNMP message to be send in such case

`EVL_WAIT_FAIL=1`
: whether or not to fail the whole workflow when the '`Run`' command fails, so when zero is set, the workflow continue regardless task failures

`EVL_WAIT_INTERVAL=2s`
: the time interval between each check for '`Wait`' command. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_WAIT_TIME=10h`
: maximal amount of time to wait for a previous '`Run`' commands to finish. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Examples

In all examples suppose an EVL project '`our_project`' represented by folder '`/home/tech_user/our_project`'.

1. Following '`ews/our_workflow.ews`':

   ```
   Run job.1.evl job.2.evl
   Run workflow.3.evl job.4.evl script.5.sh

   End
   ```

   with empty file '`workflow/our_workflow.ewf`' would be called from command line:

   ```
   cd /home/tech_user/our_project
   evl workflow/our_workflow.ewf
   ```

   which is the usual way to run workflow when testing, but running for example from a script:

   ```
   evl /home/tech_user/our_project/workflow/our_workflow.ewf
   ```

   or better:

   ```
   evl run workflow/our_workflow.ewf --project /home/tech_user/our_project
   ```

6 EVD and Data Types

'EVD' stands for 'EVL Data Definition' and it is the way how to specify structure of data sets in EVL. It can be used either inline, as a component option, or in an *.evd file.

EVL uses mostly standard C++ data types, so most of them are well known.

6.1 EVD Structure

Example first: Let's have a CSV file:

```
1;Otto Wichterle;27.10.1913;12,345.78;2025-03-19 14:34:07
2;;1.1.1970;0.00;2025-03-19 14:35:44
```

then following evd file would describe its structure:[1]

```
ID           int                 sep=";"
Name         string              sep=";"  null=""
"Birth Date" date("%-d.%-m.%Y")  sep=";"  null="1.1.1970"
Amount       decimal(12,3)       sep=";"    thousands_sep=","
"Created At" datetime            sep="\n" null="0000-00-00 00:00:00"
```

In general each nonempty line of EVD file looks like this:

```
<indent> Field_Name <blank> Data_Type <blank> EVD_Options
```

where

'<indent>'
: might be empty, 2 spaces, 4 spaces, 6 spaces, etc., to define a substructure of compound data types, see Section 6.4 [Compound Types], page 34, for details.

'Field_Name'
: is a sequence of any printable ASCII characters below 128. When a space is used, then whole field name must be quoted by double quotes. Special characters (also only ASCII ones under 128) must be escaped, e.g. '\n', '\r', '\t', '\v', '\b', '\f', '\a', '\"', '\\', or in hexa '\x??'. Characters other than letters, numbers and underscore are replaced by underscore in mappings. All these field names are valid:

```
                             // Name in mapping:
recommended_field_name       // recommended_field_name
"Field with a Space"         // Field_with_a_Space
'field-with-a-hyphen'        // _field_with_a_hyphen_
"$field_with_dollar"         // _field_with_dollar
'single_quoted'field'        // _single_quoted_field_
"with\nnewline"              // with_newline
```

'Data_Type'
: is one of:

- string, ustring, see Section 6.5 [String], page 38,
- char, uchar, short, ushort, int, uint, long, ulong, int128, uint128, see Section 6.6 [Integral Types], page 41,
- decimal, see Section 6.7 [Decimal], page 42,

[1] By setting environment variables EVL_DEFAULT_FIELD_SEPARATOR=";" and EVL_DEFAULT_RECORD_SEPARATOR=$'\n' one can avoid to use sep options.

- `float`, `double`, see Section 6.8 [Float and Double], page 43,
- `date`, `time`, `time_ns`, `interval`, `interval_ns`, `datetime`, `timestamp`, see Section 6.9 [Date and Time], page 44,

‘`EVD_Options`’
: is `<blank>` separated list of options, see Section 6.2 [EVD Options], page 32,

‘`<blank>`’
: is one or more spaces and/or tabs.

6.1.1 Comments

Standard C-style comments can be used in `evd` file, for example:

```
street_id    int
street_name  string
street_code  string  null=""  // but NOT NULL in DB
/* COMBAK: street_code will be replaced by street_num later this year
street_num   long
*/
```

6.1.2 Inline EVD

For the most of the EVL Components an inline EVD can be specified as an option. In such case comments are not allowed and the format is simply the same as for EVD in a file, just instead of newlines, commas are used to separate each field definition.

The same structure, as in above EVD Example, but as a component option (a comma separated list of fields with data types and options):

```
--data-definition='id int sep=";",
    name string sep=";" null="",
    birth_date date sep=";" null="1970-01-01",
    amount decimal(12,3) sep=";" thousands_sep=",",
    created_at datetime sep="\n" null="0000-00-00 00:00:00"'
```

6.2 EVD Options

Structure of the data is described in an EVD file – an EVL data types definition file – with file extension `.evd`.

6.2.1 Separator Definition

Field separator is defined by ‘`sep="X"`’, where ‘`X`’ can be an empty string or an ascii character below 128 specified as normal string or special character ‘`\n`’, ‘`\r`’, ‘`\t`’, ‘`\v`’, ‘`\b`’, ‘`\f`’, ‘`\a`’, ‘`\"`’, ‘`\\`’, or in hexa ‘`\x??`’ (0-7E) (as it is always a single character, ‘`\x?`’ is also possible).

Default separators can be defined:

`EVL_DEFAULT_FIELD_SEPARATOR`
: defines default field separator, when not set, `EVL_DEFAULT_FIELD_SEPARATOR='|'` is used,

`EVL_DEFAULT_RECORD_SEPARATOR`
: defines default record separator, i.e. the last field separator, when not set, `EVL_DEFAULT_RECORD_SEPARATOR='\n'` is used.

When these variables are set, then no ‘`sep=`’ options are needed in the above `EVD` example and these defaults are used instead.

> **Note:** It is recommended to use these variables only for project-wide settings in `project.sh`. Try to avoid to set them in jobs. Better use ‘`sep=`’ option in `evd` file.

In case we want to have an empty separator, for example after fixed length field, we can use '`sep=""`'.

6.2.2 Null Option

A null string by '`null="X"`' or list of strings '`null=["X","Y",...]`' can be specified. Then such string(s) will be read as '`null`' values when '`--text-input`' is used by the component.

When writing the '`null`' value by the output component with '`--text-output`' option, such string will be used instead.

When the list of null values is specified, then the first one will be used to write.

To type a special character, like newline or '`TAB`', standard hexadecimal notation can be used: '`\x??`', or also special notation for often used special characters: '`\n`', '`\r`', '`\t`', '`\v`', '`\b`', '`\f`', '`\a`', '`\"`', '`\\`'. So then to interpret a tabulator as NULL value use '`null="\t"`'.

6.2.3 Quote Option

When reading `csv` files, fields might be quoted by some character, usually by double quotes: '`"`'.

Proper parsing of such field is done by specifying attributes '`quote=`' or '`optional_quote=`'.

Specified string might be any ascii character below 128 specified as normal string or special character '`\n`', '`\r`', '`\t`', '`\v`', '`\b`', '`\f`', '`\a`', '`\"`', '`\\`', or in hexa '`\x??`' (0-7E) (as it is always a single character, '`\x?`' is also possible).

`quote="<quote_char>"`
: Use this attribute when the field is always quoted.

`optional_quote="<quote_char>"`
: Using this attribute, the field doesn't need to be quoted.

6.2.4 Encoding and Locale *(since EVL 2.5)*

`enc="<encoding>"`
: To specify an encoding of given field, string functions then behaves according to that.

`locale="<locale>"`
: To specify a locale of given field, components (like sort) then behaves according to that.

Examples

```
czech_string_in_utf8  string  enc="utf8" locale="cs_CZ"
en_string_in_utf8     string  enc="utf8" locale="en_GB"
```

When there is no encoding or locale specified in an EVD, then following environment variables can be used:

`EVL_DEFAULT_STRING_ENC=""`
: defines default encoding, when not set, empty encoding is used,

`EVL_DEFAULT_STRING_LOCALE="C"`
: defines default locale, when not set, generic '`C`' locale is used.

6.2.5 Max string length *(since EVL 2.5)*

Attributes which are used to specify maximal length of given string field. So far used only in case of load/unload tables.

`max_bytes="<number>"`
: To specify maximum Bytes of given string field. Is populated when generated based on table definition, e.g. '`VARCHAR(100 BYTES)`'.

`max_chars="<number>"`
: To specify maximum characters of given string field. Is populated when generated based on table definition, e.g. '`VARCHAR(100 CHARS)`'.

Examples:

```
string_20_bytes  string  enc="utf8" max_bytes="20"  // VARCHAR(20 BYTES)
string_20_chars  string  enc="utf8" max_chars="20"  // VARCHAR(20 CHARS)
```

Both attributes are currently used in '`Writeora`' component to know the maximal length of a string field.

6.2.6 QVD options *(since EVL 2.4)*

`qvd:format="<format_string>"`
: To specify a format string for '`timestamp`', '`datetime`', and '`date`' data types when read/write Qlik's QVD files. Example:

```
request_dt  timestamp   qvd:format="%d/%m/%Y %H:%M:%S"
some_date   date        qvd:format="%d.%m.%Y"
```

`qvd:interval`
`qvd:time`
: To be used as an attribute for '`timestamp`' and '`datetime`' data types to get an interval or time data type in Qlik's QVD files. Example:

```
request_time1  timestamp  qvd:time
request_time2  timestamp  qvd:interval
```

Compared to Qlik's time data type, interval can be larger than 24 hours. For example input timestamp '`1970-01-02 03:05:30`' would be '`03:05:30`' as time, but '`27:05:30`' as interval.

6.3 Default Values

Important

Keep in mind, that when no output record is specified in the EVM mapping (see Chapter 14 [EVM Mappings], page 174), then default value is taken, i.e. not '`nullptr`'!

data type	**default value**
`string`, `ustring`	'`""`' (empty string)
integral types, floats and decimal	'`0`' (zero)
`date`	'`1970-01-01`'
`datetime`	'`1970-01-01 00:00:00`'
`timestamp`	'`1970-01-01 00:00:00.000000000`'
`time`, `interval`	'`00:00:00`'.
`time_ns`, `interval_ns`	'`00:00:00.000000000`'.

6.4 Compound Types

`vector`
: Members can be any primitive data type or a struct type or again a vector.

`struct`
: Members can be any primitive data types or vectors or again structures.

Example of `evd` file, which defines '`vector`' and '`struct`' data types:

```
int_field          int         sep="|"
```

```
struct_field      struct     sep="|"
  double_field    double     sep=";"
  date_field1     date       sep=";"  null="1973-01-01"
vector_field      vector     sep="\n"
  short                      sep=","
datetime_field    datetime   sep=","  null="0000-00-00 00:00:00"
```

Elements of '`vector`' or '`struct`' are distinguished by indentation in `yaml` style, so by two spaces, four spaces etc.

'`struct`' and '`vector`' are especially useful for reading and writing `JSON` and `XML` files.

6.4.1 Vector

Suppose default field separator to be a pipe '`|`', i.e. `EVL_DEFAULT_FIELD_SEPARATOR="|"` and default record separator to be a newline, i.e. `EVL_DEFAULT_RECORD_SEPARATOR=$'\n'`. Then following '`evd`' file:

```
ID        int
values    vector
  int              // two spaces indentation here
```

would describe following data (in its text representation):

```
1|3;9|8|7|
2|5;11|12|13|14|15|
3|0;|
4|1;1111|
```

which would be in '`JSON`' format like this[2]:

```
[
  {"ID":1,"values":[9,8,7]},
  {"ID":2,"values":[11,12,13,14,15]},
  {"ID":3,"values":[]},
  {"ID":4,"values":[1111]}
]
```

Nicely formatted:

```
[
    {
        "ID": 1,
        "values": [
            9,
            8,
            7
        ]
    },
    {
        "ID": 2,
        "values": [
            11,
            12,
            13,
            14,
            15
```

[2] This JSON output was produced by '`evl writejson`' with option '`--array-output`'.

```
            ]
        },
        {
            "ID": 3,
            "values": [

            ]
        },
        {
            "ID": 4,
            "values": [
                1111
            ]
        }
    ]
```

Now let's suppose the values vector can be null (with its text representation '`N/A`'):

```
ID        int
values    vector null="N/A"
  int              // two spaces indentation here
```

Then following (text) data can be parsed and read with such '`evd`' file:

```
1|3;9|8|7|
2|5;11|12|13|14|15|
3|0;|
4|N/A;|
```

And in nicely formatted '`JSON`' format:

```
[
    {
        "ID": 1,
        "values": [
            9,
            8,
            7
        ]
    },
    {
        "ID": 2,
        "values": [
            11,
            12,
            13,
            14,
            15
        ]
    },
    {
        "ID": 3,
        "values": [

        ]
    },
    {
        "ID": 4,
        "values": null
    }
]
```

Mind the difference between NULL vector and zero length.

The semicolon after the number of values in text representation cannot be changed. However it worth to mention that text representation in the kind of CSV format is not usual and mostly in EVL we'd work with binary representation of the data and if we need a text representation, we'd use formats like ‘JSON’ or ‘XML’.

Complex example

And one more complex example of the ‘evd’ file with vectors. Structure of the data:

```
days      vector           null=["NULL","0"] sep="|"
  date                     null=""           sep=","
values    vector           null="NULL"       sep="|"
  struct                   null="N/A"        sep="X"
    morning decimal(12,2)  null=""           sep=","
    noon    decimal(12,2)  null=""           sep=","
    evening decimal(12,2)  null=""           sep=","
flags     vector                             sep="\n"
  uchar                                      sep=","
```

And sample of the data in nicely formatted ‘JSON’:

```
[
    {
        "days": [
            "2002-06-18",
            "2002-06-19"
        ],
        "values": [
            {
                "morning": "878.59",
                "noon": "275.69",
                "evening": "5180.64"
            },
            {
                "morning": null,
                "noon": null,
                "evening": "50.00"
            }
        ],
        "flags": [
            1,
            0
        ]
    },
    {
        "days": [
            "1999-08-01",
            "1999-09-01",
            "1999-10-01",
            "1999-11-01"
        ],
        "values": null,
        "flags": [
            0,
            0,
            0,
            0
        ]
    },
    {
        "days": [
            "2026-04-30",
            "2026-05-01",
            "2026-05-02"
        ],
        "values": [
            {
                "morning": "208.83",
                "noon": "6745.71",
                "evening": "703.54"
            },
            null,
            {
                "morning": "6519.93",
                "noon": "5220.82",
                "evening": "49.84"
            }
        ],
        "flags": [
            1,
            0,
            1
        ]
    }
]
```

Text representation is then (first and third record are split for better display):

```
2;2002-06-18,2002-06-19,|
  2;STRUCT;878.59,275.69,5180.64,X
    STRUCT;,,50.00,X|
  2;1,0,
4;1999-08-01,1999-09-01,1999-10-01,1999-11-01,|NULL;|4;0,0,0,0,
3;2026-04-30,2026-05-01,2026-05-02,|
  3;STRUCT;208.83,6745.71,703.54,X
    N/A;X
    STRUCT;6519.93,5220.82,49.84,X|
  3;1,0,1,
```

Then in mapping you can manipulate the whole vector with '`in->vector_field`'.

6.4.2 Struct

Then in mapping you can manipulate the whole structure with '`in->struct_field`'. Particular element of '`struct`' you can then reach by '`in->struct_field->double_field`' for example.

6.5 String

Standard C++ library '`std::basic_string`' is used for strings. For details see

http://en.cppreference.com/w/cpp/string/basic_string

`string`	size: up to 2^{64} Bytes (i.e. limited only by memory)

An EVD file Example:

```
field_name1  string(10)
field_name2  string(10) sep=""
field_name3  string     sep=";" null="NULL"
field_name4  string             null=""              quote="\""
field_name5  string             null=["","N/A","NA"]
last_field   string
```

where

‘`field_name1`’
: cannot be NULL and has fixed length 10 bytes, followed by the value of `$EVL_DEFAULT_FIELD_SEPARATOR` environment variable.

‘`field_name2`’
: cannot be NULL and has fixed length 10 bytes, with no separator.

‘`field_name3`’
: is nullable and string ‘`NULL`’ is interpreted as NULL value. End of the field is represented by character ‘`;`’.

‘`field_name4`’
: is nullable and empty string is interpreted as NULL value. Field is quoted by ‘`"`’, but for an empty string, quotes are not needed. The end of the field is represented by `$EVL_DEFAULT_FIELD_SEPARATOR`.

‘`field_name5`’
: is nullable and empty string, ‘`N/A`’ and ‘`NA`’ are interpreted as NULL value when reading, but when writing into text file, NULL is represented by the first one, i.e. an empty string. The end of the field is represented by `$EVL_DEFAULT_FIELD_SEPARATOR`.

‘`last_field`’
: cannot be NULL and the end of the field is represented by `$EVL_DEFAULT_RECORD_SEPARATOR`.

Example of four records which can be parsed by above EVD file definition.

```
          |          NULL;"second string field"|NA|last field
0123456789|0123456789first string field;""|N/A|last field
----------|----------;"  ;  second field  |  "|third string field|last field
abcdefghij|abcdefghij       ;||last field
```

Neither `EVL_DEFAULT_FIELD_SEPARATOR` nor `EVL_DEFAULT_RECORD_SEPARATOR` is set, so default values are used, i.e. ‘`|`’ and ‘`\n`’.

6.5.1 Manipulation

Standard methods from the library ‘`basic_string`’, where ‘`a`’, ‘`b`’ are strings, ‘`c`’ is a char, ‘`i`’ is an (unsigned) int):

`a.empty()`
: checks whether the string ‘`a`’ is empty,

`a.size(), a.length()`
: returns the number of characters,

`a.clear()`
: clears the contents of string ‘`a`’,

`a.insert()`
: inserts characters,

`a.erase(position,size)`
: removes from string ‘`a`’ characters from after ‘`position`’ of the size ‘`size`’,

`a.push_back(c)`
: appends a character ‘`c`’ to the end,

`a.pop_back()`
: removes the last character,

`a.append(b), +=`
: appends characters to the end,

`operator +`
: concatenates two strings or a string and a char,

`a.replace(position,size,b)`
: replaces in string '`a`' from after '`position`' of size '`size`' by string '`b`',

`a.substr(position,size)`
: returns a substring of '`a`' from after '`position`' and of length '`size`',

`a.copy(b)`
: copies characters,

`a.resize(i,c)`
: changes the number of characters stored, if '`i`' is shorter than current length then it simply cuts, if '`i`' is longer, then add character '`c`' to fill the length '`i`',

`a.swap(b)`
: swaps the contents of '`a`' and '`b`'.

6.5.2 Search

`find()` find characters in the string,

`rfind()` find the last occurrence of a substring,

`find_first_of()`
: find first occurrence of characters,

`find_first_not_of()`
: find first absence of characters,

`find_last_of()`
: find last occurrence of characters,

`find_last_not_of()`
: find last absence of characters.

6.5.3 Comparison

`Operators ==, != <, >, <=, >=`
: lexicographically compares two strings,

`compare()`
: compares two strings.

6.5.4 Numeric conversions

`stoi()` converts a string to an integer,

`stol()` converts a string to a long,

`stoul()` converts a string to an unsigned long,

`stof()` converts a string to a float,

`stod()` converts a string to a double,

`to_string()`
: converts an integral or floating point value to string.

6.5.5 EVL specific string functions

The advantage of using EVL specific function is that they handle NULLs, i.e. when the string is NULL, then also the output is NULL. Using native C++ functions need to handle NULLs conditionally.

The list of such function:

`hex_to_str(str), str_to_hex(str)`
: to convert ordinary string to its hexadecimal representation and vice versa,

`length(str)`
: returns the length of given string,

`md5sum(str)`
: returns MD5 checksum,

`sha256sum(str), ...`
: SHA checksum functions,

`split(str,char)`
: to split a string into a vector,

`starts_with(str,substr), ends_with(str,substr)`
: to check if a string starts or ends with a given character or string '`substr`',

`str_compress(str,method), str_uncompress(str,method)`
: to un/compress a string by given method: snappy or gzip,

`str_count(str,substr)`
: returns the number of '`substr`' occurrences,

`str_index(str,substr), str_rindex(str,substr)`
: returns the position of '`substr`' from left or right,

`str_mask_left(str,len,char), str_mask_right(str,len,char)`
: to replace by '`char`' the specified number of characters from left/right,

`str_pad_left(str,len,char), str_pad_right(str,len,char)`
: to add from left/right the specified character, up to the given length,

`str_replace(str,strA,strB)`
: to replace a string or character '`strA`' by '`strB`',

`substr(str,pos,len)`
: it returns a substring starting after position '`pos`' of the specified length '`len`'.

`trim(str), trim_left(str), trim_right(str)`
: to trim a string by specified character,

`uppercase(str), lowercase(str)`
: to change to uppercase or lowercase string, ...

where '`str`' is the string or a pointer to the string.

See Section 14.2 [String Functions], page 176, for details.

6.6 Integral Types

All integral data types are standard C++ ones.

`char`	size: 1 Byte, min: −128, max: 127
`uchar`	size: 1 Byte, min: 0, max: 255

`short`	size: 2 Bytes, min: −32 768, max: 32 767
`ushort`	size: 2 Bytes, min: 0, max: 65 535
`int`	size: 4 Bytes, min: −2 147 483 648, max: 2 147 483 647
`uint`	size: 4 Bytes, min: 0, max: 4 294 967 295
`long`	size: 8 Bytes, min: -2^{63} (approx. -9×10^{18}), max: $2^{63}-1$ (approx. 9×10^{18})
`ulong`	size: 8 Bytes, min: 0, max: $2^{64}-1$ (approx. 18×10^{18})
`int128`	size: 16 Bytes, min: -2^{127} (approx. -1.7×10^{38}), max: $2^{127}-1$ (approx. 1.7×10^{38})
`uint128`	size: 16 Bytes, min: 0, max: $2^{128}-1$ (approx. 3.4×10^{38})

Except '`sep=`', '`null=`', '`quote=`', '`optional_quote=`', no other options are possible for these data types.

6.7 Decimal

Decimal data type is defined by '`decimal(m,n)`', where '`m`' is number of all digits and '`n`' is the number of decimal places. Decimal is EVL custom data type.

`decimal(m,n)`
: when '`n`' is missing, zero is supposed
: size: 8 Bytes for '`m`' up to 18 digits
: size: 16 Bytes for '`m`' from 19 to 38 digits

Next to standard EVD options (i.e. '`sep=`', '`null=`', '`quote=`', '`optional_quote=`') decimal and thousands separator can be specified:

`decimal_sep="."`
: to specify a decimal separator, which can be any single ascii character below 128; by default it is a decimal point

`thousands_sep=""`
: defines how to separate thousands, it can be any single ascii character below 128; by default there is no thousands separator.

An EVD file example:

```
revenues  decimal(9,4)  decimal_sep="," thousands_sep="."  // e.g. 12.345,6789
expenses  decimal(18)                        // e.g. 123456789012345678
taxes     decimal(18,6) thousands_sep=" "    // e.g. 123 456 789 012.345678
latitude  decimal(10,6)                      // e.g. 49.8197203
longitude decimal(10,6) decimal_sep=","      // e.g. 18,1673552
```

6.7.1 Declaration in mapping

Object creation:

```
decimal d();           // 0       no decimal places
decimal d(821);        // 821     initialization from int, no dec. places
decimal d(821, 3);     // 821.000 initialization from int, 3 dec. places
decimal d(821.658, 3); // 821.658 init. from float/double, 3 dec. places
decimal d2(d, 2);      // 821.65  initialization from existing object,
                       //         just change decimal places to 2 (cut off)
```

6.7.2 Manipulation, comparison

Increment/decrement:

```
d++;
++d;
d--;
--d;
```

All following operations can be done between two decimals or between decimal and any integral data type:

```
d += 100;               //   921.65
d -= decimal(0.66, 3); //   920.990
d *= -2                 // -1841.980
d /= 2                  //  -920.990
decimal d2 = d + 120;   //  -800.990
```

When adding, subtracting or dividing, the result has higher decimal places from both operands. When multiplying two decimals, the decimal places are added.

```
1.23 + 6.0000 =  7.2300
1.23 - 6.0000 = -4.7700
1.23 * 6.0000 =  7.380000
1.23 * 6.0000 =  0.2050
```

Comparison as usual: '`==`', '`!=`', '`<`', '`>`', '`<=`', '`>=`'.

```
if (d > 128) ...
if (d == decimal(123.456, 3)) ...
if (d <= d2) ...
```

The decimal places can be obtained and set by the following tho methods.

```
d.scale()       // 3
d.set_scale(2)  // set the scale to 2
```

These methods convert decimal to other data types

```
int i      = d.to_int();    // cut off fractional part
float f1   = d.to_float();
double f2  = d.to_double();
string str = d.to_string('.', ''); // decimal_separator = '.'
                                   // thousand_separator = ''
string str = d.to_string();        // use default separators
```

Important: By any operation, when the precision is decreased, there is no rounding, just cut off!

The reason is performance, to rounding the number, use '`round()`' method:

```
decimal costs(856.128, 3);              // 856.128
decimal costs_rounded = costs.round(2); // 856.130
decimal costs_cut_off(costs, 2);        // 856.12
```

6.8 Float and Double

Float and double are standard C++ data types.

`float`	size: 4 Bytes, range: $\pm$ 3.4 $\times 10^{\pm 38}$ (about 7 digits)
`double`	size: 8 Bytes, range: $\pm$ 1.7 $\times 10^{\pm 308}$ (about 15 digits)

Except '`sep=`', '`null=`', '`quote=`', '`optional_quote=`', no other options are possible for these data types.

> **Note:** Compared to `decimal(m,n)` data type, operating with floats and doubles (doing summations for example), usually leads to approximated values. So it is usually good idea to avoid using these data types for money and such.

Example

With EVD file

```
sent_mb      float  sep="|"  null=""
received_mb  float  sep="\n" null=""
```

you can read source `csv` file like this:

```
0.321e12|1.234E-02
12.78E11|3.798
```

6.9 Date and Time

Date, time, time_ns, interval, interval_ns and datetime are data types based on standard C++ library '`std::time`'. Timestamp is built upon Google's '`cctz`' library.

`date` *(since EVL 1.0)*

to store a date, i.e. day, month and year
size: 4 Bytes, range: 1970-01-01 ± approx. 6×10^{11} years
first 2 Bytes keeps a year, then 1 Byte for month and 1 Byte for day
example: `2008-04-20`

`time` *(since EVL 2.8)*

to store a day time, i.e. hour, minute and second
size: 4 Bytes, range: 00:00:00 – 23:59:59
example: `13:35:00`

`time_ns` *(since EVL 2.8)*

to store a day time with nanoseconds
size: 8 Bytes, range: 00:00:00.000000000 – 23:59:59.999999999
example: `13:37:00.350000000`

`interval` *(since EVL 2.8)*

to store a time interval in hours, minutes and seconds
size: 4 Bytes, min: 00:00:00
example: `165:35:00`

`interval_ns` *(since EVL 2.8)*

to store a time interval with nanoseconds
size: 8 Bytes, min: 00:00:00.000000000
example: `165:35:00.123456789`

`datetime` *(since EVL 1.0 as timestamp, since EVL 2.4 as datetime)*

to store a date and time, i.e. year, month, day, hour, minute and second
size: 8 Bytes, range: 1970-01-01 00:00:00 ± approx. 6×10^{11} years
example: `2010-07-01 09:02:00`

`timestamp` *(since EVL 2.4)*

to store a date and time with nanoseconds and with a time zone, i.e. year, month,

day, hour, minute, second, nanoseconds and possibly a time zone
size: 12 Bytes, range: 1970-01-01 00:00:00 ± approx. 6 $\times 10^{11}$ years
example: `2015-05-09 13:37:00.000 +02:00`

6.9.1 Format string

As an argument (in curly brackets) formatting pattern can be specified. Standard C notation is used.

When no argument to date and time data types are provided, defaults are used:

`EVL_DEFAULT_DATE_PATTERN`
to specify default formatting string for '`date`' data type,
by default it is `"%Y-%m-%d"`

`EVL_DEFAULT_TIME_PATTERN`
to specify default formatting string for '`time`' data type,
by default it is `"%H:%M:%S"`

`EVL_DEFAULT_DATETIME_PATTERN`
to specify default formatting string for '`datetime`' data type,
by default it is `"%Y-%m-%d %H:%M:%S"`

`EVL_DEFAULT_TIMESTAMP_PATTERN`
to specify default formatting string for '`timestamp`' data type,
by default it is `"%Y-%m-%d %H:%M:%E*S"`

All possible format strings:

`%%`	a literal '`%`'
`%a`	locale's abbreviated weekday name (e.g. '`Sun`')
`%A`	locale's full weekday name (e.g. '`Sunday`')
`%b`	locale's abbreviated month name (e.g. '`Jan`')
`%B`	locale's full month name (e.g. '`January`')
`%c`	locale's date and time (e.g. '`Thu Mar 3 23:05:25 2005`')
`%C`	century; like '`%Y`', except omit last two digits (e.g. '`20`')
`%d`	day of month (e.g. '`01`')
`%D`	date; same as '`%m/%d/%y`'
`%e`	day of month, space padded; same as '`%_d`'
`%Ez`	RFC3339-compatible numeric UTC offset (+hh:mm or -hh:mm)
`%E*z`	full-resolution numeric UTC offset (+hh:mm:ss or -hh:mm:ss)
`%E#S`	seconds with # digits of fractional precision
`%E*S`	seconds with full fractional precision (a literal '*')
`%E#f`	fractional seconds with # digits of precision
`%E*f`	fractional seconds with full precision (a literal '*')
`%E4Y`	four-character years (-999 ... -001, 0000, 0001 ... 9999)
`%ET`	the RFC3339 "date-time" separator "T"
`%F`	full date; same as '`%Y-%m-%d`'

`%g`	last two digits of year of ISO week number (see '`%G`')
`%G`	year of ISO week number (see '`%V`'); normally useful only with '`%V`'
`%h`	same as '`%b`'
`%H`	hour ('`00`'..'`23`')
`%I`	hour ('`01`'..'`12`')
`%j`	day of year ('`001`'..'`366`')
`%k`	hour, space padded ('` 0`'..'`23`'); same as '`%_H`'
`%l`	hour, space padded ('` 1`'..'`12`'); same as '`%_I`'
`%m`	month ('`01`'..'`12`')
`%M`	minute ('`00`'..'`59`')
`%n`	a newline
`%p`	locale's equivalent of either '`AM`' or '`PM`'; blank if not known
`%P`	like '`%p`', but lower case
`%r`	locale's 12-hour clock time (e.g. '`11:11:04 PM`')
`%R`	24-hour hour and minute; same as '`%H:%M`'
`%s`	seconds since '`1970-01-01 00:00:00 UTC`'
`%S`	second ('`00`'..'`60`')
`%t`	a tab
`%T`	time; same as '`%H:%M:%S`'
`%u`	day of week ('`1`'..'`7`'); '`1`' is Monday
`%U`	week number of year, with Sunday as first day of week ('`00`'..'`53`')
`%V`	ISO week number, with Monday as first day of week ('`01`'..'`53`')
`%w`	day of week ('`0`'..'`6`'); '`0`' is Sunday
`%W`	week number of year, with Monday as first day of week ('`00`'..'`53`')
`%x`	locale's date representation (e.g. '`12/31/99`')
`%X`	locale's time representation (e.g. '`23:13:48`')
`%y`	last two digits of year ('`00`'..'`99`')
`%Y`	year
`%z`	+hhmm numeric time zone (e.g., -0400)
`%Z`	alphabetic time zone abbreviation (e.g., EDT)

By default, date pads numeric fields with zeroes. The following optional flags may follow '`%`'.

`-`	(hyphen) do not pad the field
`_`	(underscore) pad with spaces
`0`	(zero) pad with zeros
`^`	use upper case if possible
`#`	use opposite case if possible

6.9.2 EVD Example

Following dates definition are equivalent.

```
valid_from  date
valid_from  date("%F")
valid_from  date("%Y-%m-%d")
```

Following datetimes are all the same.

```
request_dt  datetime
request_dt  datetime("%F %T")
request_dt  datetime("%Y-%m-%d %H:%M:%S")
```

Following timestamps are all the same.

```
request_dt  timestamp
request_dt  timestamp("%F %T.%E9f")
request_dt  timestamp("%Y-%m-%d %H:%M:%S.%E9f")
```

QVD's format string can be specified:

```
request_dt  timestamp  qvd:format="%d/%m/%Y %H:%M:%S"
some_date   date       qvd:format="%d.%m.%Y"
```

6.9.3 Qlik's time

When time need to be specified in QVD file, then standard timestamp need to be provided, just with '`qvd:time`' option. Then the date is simply cut off from the timestamp to be stored in QVD:

```
request_time  timestamp("%H:%M:%S")  qvd:time
```

6.9.4 Qlik's interval

When interval data type need to be specified in QVD file, then standard timestamp need to be provided, just with '`qvd:interval`' option. Then the time is taken since '`1970-01-01`':

```
request_time  timestamp("%Y-%m-%d %H:%M:%S")  qvd:interval
```

Note: Compared to Qlik's time data type, interval can be larger than 24 hours. For example input timestamp '`1970-01-02 03:05:30`' would be '`03:05:30`' as time, but '`27:05:30`' as interval.

6.9.5 Declaration in mapping

The following declarations are equivalent.

```
static datetime min_date(1970,1,1,0,0,0);
static datetime min_date("1970-01-01 00:00:00");
static datetime min_date("1970-01-01 00:00:00", "%Y-%m-%d %H:%M:%S");
static datetime min_date = datetime::from_epoch_time(0);

static date min_date(1970,1,1);
static date min_date("1970-01-01");
static date min_date("1970-01-01", "%Y-%m-%d");
```

6.9.6 Manipulation, comparison

Lets have '`datetime dt(2017,5,31,19,37,0)`' and '`date d(2018,1,14)`' in the following examples.

Methods switching date and datetime data type:

`dt.to_date()`
: returns '`2017-05-31`', i.e. cut off time and return date data type

`dt.to_datetime()`
returns '2018-01-14 00:00:00', i.e. add '00:00:00' and return datetime data type

Following methods return appropriate values as '`int`'.

```
dt.year()       -- 2017
 d.year()       -- 2018
dt.month()      -- 5
 d.month()      -- 1
dt.day()        -- 31
 d.day()        -- 14
dt.hour()       -- 19
dt.minute()     -- 37
dt.second()     -- 0
dt.epoch_time() -- 1496169420
dt.yearday()    -- 151
 d.yearday()    -- 14
dt.weekday()    -- 3 (Wednesday)
 d.weekday()    -- 0 (Sunday)
```

In the context of string, method '`weekday()`' returns '`sunday`', . . .

'`weekday()`' returns '`0`' for Sunday, '`1`' for Monday, . . ., '`6`' for Saturday.

These methods convert date and datetime to string:

```
string str1 = dt.to_string();     // 2017-05-31 19:37:00,
                                  // i.e. uses default format string
string str2 = dt.to_string("%Y%m%d%H");     // 2017053119
string str3 = min_date.to_string("%Y%m%d"); // 19700101
```

Comparison: '`==`', '`!=`', '`<`', '`>`', '`<=`', '`>=`'.

```
if (dt >= datetime(1990,1,1) { ... }
```

Addition, subtraction:

```
dt += 65;          // add 65 seconds, i.e. 2017-05-31 19:38:05
dt--;              // 2017-05-31 19:38:04
date d(2017,5,31);
d -= 35;           // subtract 35 days, i.e. 2017-04-26

dt.add_year(1);    // 2018-05-31 19:38:04
 d.add_month(-1);  // 2017-03-26
dt.add_day(6);     // 2018-06-06 19:38:04
dt.add_hour(-2);   // 2018-05-31 17:38:04
dt.add_minute(3);  // 2018-05-31 19:41:04
dt.add_second(-6); // 2018-05-31 19:37:58
```

The difference between '`dt.add_second(10)`' and '`dt+10`' is that in the first case we modify the object itself, but in the second case new value is returned. One can use then for example '`dt.add_hour(2).add_minute(3)`'.

Difference:

```
auto diff = dt - datetime(2018,5,31,19,36,57); // 61 (seconds)
auto diff =  d - date("2017-04-02");           // -6 (days)
```

Let's summarize the logic:

```
date - int  => date          datetime - int      => datetime
date - date => int           datetime - datetime => int
```

7 Components Common

ETL jobs are defined in `evs` files and consist of components connected by flows.

In this chapter, all components are described.

There are two kinds of component commands:

- '`Component`', i.e. component name begins with capital letter – these are intended to be used in `evs` file (EVL job structure files),
- '`evl component`', i.e. command '`evl`' with the first argument to be a name of the component – these invocations are to be used as standalone commands, which can be run from commandline.

7.1 Common Options

Mostly this invocation schema is applied on all components.

```
<Component>  F_IN... F_OUT... EVD_IN... EVD_OUT... EVM
        [ -v|--validate ]
        [ -x|--text-input ]
        [ -y|--text-output ]
```

where `<Component>` is the name of the component, like `Read` or `Join`.

`F_IN... F_OUT...`
: are names of the input flows or files and of the output flows or files.

- **Flow names** must be unique in each `evs` file. Convention is to have these names in CAPITAL LETTERS.
- **Files** – if `F_IN` or `F_OUT` contains '`/`' (foreslash), then it is treated as a file, not a flow. Remember that one can use special files, like `/dev/stderr` or `/dev/null` or in general `/dev/fd/N`.

`EVD_IN... EVD_OUT...`
: specifies the `evd` with the information about data structure, i.e. field names, data types, nullability, separator, etc. It can be:

- **EVD file** – a file with the data structure definition;
- **Inline EVD** – evd can be specified inline by options this way:

  `-D | --input-definition`
  : for input evd (when both – input and output – evd have to be specified);

  `-d | --output-definition`
  : for output evd (when both – input and output – evd have to be specified);

  `-d | --data-definition`
  : for input evd (whenonly one evd is needed by `<Component>`).

`EVM`
: specifies the `evm` file which contains the mapping of the component.

All options are interpreted as in shell, so must be specified in one line or separated by '\' at the end of the line.

8 Basic Components

Most of these basic components follows standard GNU/Linux commands, their purpose is obvious immediately.

Standard ETL components

8.1 Assign *(since EVL 1.2)*

Assign the content of input flow or file `<f_in>` into shell variable `<varname>`, which is then exported into environment. Don't forget to apply '`--text-output`' on preceding component to get text content in the `<variable>`.

This component doesn't work for partitioned flow.

`Assign`

: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

There is no standalone version of this component as you can use standard Bash behaviour for this purpose. For example:

```
VARNAME=$(evl cat filename some.evd --text-output)
```

EVS is EVL job structure definition file, for details see evl-evs(5).

Synopsis

```
Assign
  <f_in> <varname>

evl assign
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. EVL job (an ‘`evs`’ file) which reads content of a binary file ‘`hwm.bin`’ into variable ‘`HWM`’:

   ```
   Read    hwm.bin   FLOW_HWM  evd/some.evd  --text-output
   Assign  FLOW_HWM  HWM
   ```

   Such a value can be then used (after ‘`Wait`’ component!) within mapping by:

   ```
   static int hwm = getenv_int("HWM",0);   // use 0 when $HWM is empty
   *out->incremental_id = ++hwm;
   ```

2. To get a value from text file:

   ```
   Assign  hwm.txt  HWM
   ```

3. To assign flow content into a ‘`NATCO`’ variable:

   ```
   Map     FLOW_01  FLOW_02 in.evd out.evd map.evm  --text-output
   Assign  FLOW_02  NATCO
   ```

8.2 Cat *(since EVL 1.0)*

Concatenate flows or files.

`Cat`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl cat`
: is intended for standalone usage, i.e. to be invoked from command line.

EVD is EVL data definition file, for details see ‘`man 5 evd`’.

Synopsis

```
Cat
  <f_in>... <f_out> (<evd>|-d <inline_evd>)
  [ --validate ]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]
```

```
evl cat
  [<file>...]  (<evd>|-d <inline_evd>)
  [ --validate ]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]
  [ -v|--verbose ]

evl cat
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, user_id string enc=iso-8859-1'`'

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`--text-input-dos-eol`
: suppose the input as text with CRLF as end of line

`--text-input-mac-eol`
: suppose the input as text with CR as end of line

`-y, --text-output`
: write the output as text, not binary

`--text-output-dos-eol`
: produce the output as text with CRLF as end of line

`--text-output-mac-eol`
: produce the output as text with CR as end of line

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

Print to stdout binary input in text format:

```
evl cat example.evd -y <input.bin
```

8.3 Cmd *(since EVL 1.2)*

Basicly it calls:

```
cat <f_in> | <command> > <f_out>
```

When `<f_in>` is empty, then it runs:

```
<command> > <f_out>
```

and when `<f_out>` is empty:

```
cat <f_in> | <command>
```

`<command>` can be also a pipeline.

If `<f_in>` is partitioned, then `<command>` is applied on all partitions and keep the output `<f_out>` also partitioned.

Synopsis

```
Cmd
  <f_in> <f_out> <command>

evl cmd
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Write 10 times 'repeat some error message' to the STDERR and into EVL job log:

   ```
   Cmd "" /dev/stderr "yes repeat some error message | head"
   ```

2. Suppose from '`SOME_FLOW`' we obtain integers, one by line, then median can be obtained from R and be written into '`/some/file`':

   ```
   Cmd SOME_FLOW /some/file "Rscript median.R"
   ```

   The file median.R might look like this:

   ```
   f <- file('stdin'); open(f); x <- c();
   while ( length( line <- readLines(f) ) > 0 ) x <- c(x,as.integer(line));
   write(median(x), stdout());
   ```

8.4 Component *(since EVL 1.0)*

Run `<component>` from the project's evc directory with arguments `<comp_arg>`. In the `<component>` these arguments are available as the array '`COMP_ARG[1]`', '`COMP_ARG[2]`', ... '`COMP_ARG[0]`' is the component's name.

When the `<component>` is not in current project subdirectory '`evc/`', it tries the folder '`$EVL_EVC_DIR/`'.

You can also specify the full path to the component. Check examples.

Flow names within a component have unique prefixes, so cannot be in conflict with those in the job. However if you need to connect output flow(s) of the component, you need to use variable '`$COMP_FLOW`' which is set by the component to such a prefix. So then flow from the component, e.g. '`FLOW_IN_COMP`', can be read in parent job as '`$COMP_FLOW.FLOW_IN_COMP`'. Check examples.

For input flow there is a variable '`$PARENT_FLOW`' which can be used in the component. Parent flow '`FLOW_INTO_COMP`' can be reference within a component as '`$PARENT_FLOW.FLOW_INTO_COMP`'. Check examples for better understanding.

`Comp`

  is to be used in EVS job structure definition file.

`evl comp`

  is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVS is EVL job structure definition file, for details see evl-evs(5).

Synopsis

```
Comp
  <component> [<comp_arg>...]

evl comp
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`

  print this help and exit

`--usage`

  print short usage information and exit

`--version`

  print version and exit

Examples

1. Run custom component '`evc/prepare_lkp.evc`' with neither input nor output:

   ```
   Comp prepare_lkp
   ```

2. Run component from EVL Data Hub template project with three arguments:

   ```
   Comp $EVL_TEMPLATE_DIR/data-hub/evc/scd2_read_increments.evc party.*.csv evd/party.
   ```

3. Reading output from the component. Suppose you have custom generic component ‘`evc/read_files.evc`’ which do some magic with json files, e.g.:

   ```
   jsons="${COMP_ARG[1]}"
   evd="${COMP_ARG[2]}"
   key="${COMP_ARG[3]}"
   Read "$jsons" JSONS "$evd"
   Tee  JSONS A B "$evd" --key="$key"
   ```

   And you need to connect these output flows ‘`A`’ and ‘`B`’ into your job, e.g.:

   ```
   Comp read_files /landing/users.*.json evd/users.evd surname
   Sort  $COMP_FLOW.A SORTED evd/users.evd
   Write $COMP_FLOW.B users.csv evd/users.evd
   ...
   ```
4. Writing flow to the component. Suppose you have custom component ‘`evc/write_log.evc`’, e.g.:

   ```
   flow_in="${COMP_ARG[1]}"
   Write $flow_in some_file.log -d "X string" --text-output
   ```

   In the job it would look like this:

   ```
   Tail XXX LOG evd/XXX.evd -n 100
   Comp write_log.evc LOG
   ```

   Alternatively the component would look like this as well:

   ```
   Write $PARENT_FLOW.LOG some_file.log -d "X string" --text-output
   ```

8.5 Cut *(since EVL 1.0)*

Remove columns from input records. Use this component when you want to reduce the number of columns.

`Cut`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl cut`
: is intended for standalone usage, i.e. to be invoked from command line and read records from standard input and write to standard output.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Cut
  <f_in> <f_out> (<evd_in>|-D <inline_evd) (<evd_out>|-d <inline_evd>)
  [--validate] [-x|--text-input] [-y|--text-output]

evl cut
  (<evd_in>|-D <inline_evd) (<evd_out>|-d <inline_evd>)
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl cut
  ( --help | --usage | --version )
```

Options

`-D, --input-definition=<inline_evd>`
: either this option or the file `<evd_in>` must be presented. Example: -D 'id int, user_id string'

`-d, --output-definition=<inline_evd>`
: either this option or the file `<evd_out>` must be presented. Example: -d 'user_sum long'

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Print to stdout only integer field '`id`':

```
evl cut example.evd -d'id int' -xy <in.txt
```

8.6 Departition *(since EVL 1.2)*

Gather or merge partitions into one output flow or file. When '`-k <key>`' is specified, then sorted input of each partition is supposed and output will be again sorted (i.e. merged). With no '`-k <key>`', it gather input partitions in round-robin fashion. Applying to only one partition simply write input to output. EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Departition
  <f_in>... <f_out> (<evd>|-d <inline_evd>)
  (--key=<key> | --round-robin)
  [--validate] [-x|--text-input] [-y|--text-output]

evl departition
  <file_in> <file_out> (<evd>|-d <inline_evd>)
  (--key=<key> | --round-robin)
  [-v|--validate] [-x|--text-input] [-y|--text-output]
```

```
    [-v|--verbose]

  evl departition
    ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-k, --key=<key>`
: merge partitioned flows/files according to the key, so the output is sorted by this key

`-r, --round-robin`
: gather in round-robin fashion

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To departition partitioned flow in the EVL job:

```
Read  gs://my_bucket/cust.csv CUST $EVD_CUST
Partition   CUST      CUST_P  $EVD_CUST --round-robin
Map         CUST_P    PROC_M  $EVD_CUST $EVD_PROC $EVM_PROC
Departition PROC_M    PROC_G  $EVD_PROC --round-robin
Write       PROC_G    gdrive://proc.xlsx $EVD_PROC
```

8.7 Echo *(since EVL 2.0)*

Write `<string>` into `<f_out>`. This component doesn't produce partitioned flow.

'`Echo`' is to be used in EVS job structure definition file.

`<f_out>` is either output file or flow name.

There is no standalone version of this component as you can use standard '`echo`'.

EVS is EVL job structure definition file, for details see evl-evs(5).

Synopsis

```
Echo
  <string> <f_out> [-e] [-n]

evl echo
  ( --help | --usage | --version )
```

Options

`-n`
: do not output the trailing newline (standard Bash echo option)

`-e`
: enable interpretation of backslash escapes (standard Bash echo option)

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. An EVL job (specified in '`evs`' file) which run simple select statement from Postgreql table:

   ```
   Echo   "select max(id) from some_db.some_table;" SELECT
   RunPG  SELECT MAX_ID
   ```

2. To add two hardcoded records to the end of a flow:

   ```
   ...    ...    FLOW    -d "s string"
   Echo  "Some string footer,\nwith two lines." FOOTER -e
   Cat   FLOW   FOOTER  -d "s string"
   ...
   ```

8.8 Filter *(since EVL 1.0)*

Filter records by the `<condition>`. Records for which the `<condition>` is false, are forwarded to a reject file or to a flow if specified.

In many cases filtering records would be better to do in '`Map`' component using '`discard()`' function. Having '`Filter`' component right before or after a '`Map`' is not perfomance optimal. Check '`man evl-map`' for details.

Also using '`Filter`' right after a '`Read`' component is usually not performance optimal. It is usually better to shift filtering to the database for example. Check option '`--where`' of '`Read`' component for details.

`Filter`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl filter`
: is intended for standalone usage, i.e. to be invoked from command line and read records from standard input and write to standard output.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Filter
  <f_in> <f_out> (<evd>|-d <inline_evd>) <condition>
  [-r|--reject=<f_out>]
  [-x|--text-input] [-y|--text-output]

evl filter
  (<evd>|-d <inline_evd>) <condition>
  [-r|--reject=<f_out>]
  [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl filter
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

-r, –reject=<f_out> catch rejected records into file or flow.

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

Command line invocation examples:

1. To print to stdout only records from file '`ID.txt`' with value of id less than 100:

```
evl filter -d 'id int' -xy '*id<100' < ID.txt
```

Field '`id`' is a pointer, so to get the value, '`*id`' must be used.

2. Print to stdout only records from file '`IDs.csv`' where '`id1`' is different from '`id2`', records with the same ids will be send into '`same_IDs.csv`':

```
evl filter -d 'id1 int sep=",", id2 int' -xy -r same_IDs.csv \
    '*id1 != *id2' < IDs.csv
```

EVL job examples:

3. In an ‘evs’ file:

```
...     ...    SOURCE  evd/sample.evd
Filter  SOURCE OUTPUT  evd/sample.evd  "price && *currency == \"EUR\""
...     OUTPUT ...     evd/sample.evd
```

This example filter out records with NULL ‘price’ and with currency other than ‘EUR’. (‘price’ is a pointer, so simply specifying ‘price’ in the condition means ‘price != nullptr’.)

4. If there would be a ‘Read’ component right before the ‘Filter’, then consider using option ‘--where’ instead, because in such case the filter is shifted to the source DB, e.g.:

```
SRC_HOST_URI="postgres://tech_etl@pg_server:5432"
SRC_PATH="dwh_db?schema=public&table=invoices"

Read    $SRC_HOST_URI/$SRC_PATH INVOICES_EUR evd/invoices.evd \
            --where "price is not null AND currency = 'EUR'"
Map     INVOICES_EUR            EUR_MAP      evd/invoices.evd ...
```

will run the query in PostgreSQL database with where condition:

```
WHERE price is not null AND currency = 'EUR'
```

One can also use EVL notation with this ‘--where’ option, e.g.:

```
SRC_HOST_URI="postgres://tech_etl@pg_server:5432"
SRC_PATH="dwh_db?schema=public&table=invoices"

Read    $SRC_HOST_URI/$SRC_PATH INVOICES_EUR evd/invoices.evd \
            --where 'price && *currency == "EUR"'
Map     INVOICES_EUR            EUR_MAP      evd/invoices.evd ...
```

so then it would work also in case of reading a file:

```
Read    data/invoices.csv INVOICES_EUR evd/invoices.evd \
            --where 'price && *currency == "EUR"'
Map     INVOICES_EUR      EUR_MAP      evd/invoices.evd ...
```

in such case it is then internally the same as:

```
Read    data/invoices.csv INVOICES_SRC evd/invoices.evd
Filter  INVOICES_SRC      INVOICES_EUR evd/invoices.evd \
            'price && *currency == "EUR"'
Map     INVOICES_EUR      EUR_MAP      evd/invoices.evd ...
```

5. And using ‘Filter’ to split a flow:

```
...     ...    INV  evd/invoices.evd
Filter  INV    EUR  evd/invoices.evd -r NONEUR '*currency == "EUR"'
Sort    EUR    EUR_SRT    evd/invoices.evd --key "price"
Sort    NONEUR NONEUR_SRT evd/invoices.evd --key "currency,price"
...
```

8.9 Gather *(since EVL 1.2)*

Gather several input flows or files into one output flow or file in round-robin fashion.

`Gather`

is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl gather`
: is intended for standalone usage, i.e. to be invoked from command line. When `<file>` is '-', then read from stdin.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Gather
  <f_in>... <f_out> (<evd>|-d <inline_evd>)
  [--validate] [-x|--text-input] [-y|--text-output]

evl gather
  [<file>...]  (<evd>|-d <inline_evd>)
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl gather
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Following command:
   ```
   evl gather file.a file.b file.c file.evd -xy
   ```
   print to stdout first record of '`file.a`' then first record of '`file.b`' then first record of '`file.c`', then second records and so on
2. To gather partitioned flow in the EVL job:
   ```
   Read      s3://my_bucket/cust.csv CUSTOMERS $EVD_CUST
   ```

```
Partition CUSTOMERS CUST_P  $EVD_CUST --round-robin
Map       CUST_P    PROC_M  $EVD_CUST $EVD_PROC $EVM_PROC
Gather    PROC_M    PROC_G  $EVD_PROC
Write     PROC_G    sftp:///some/path/proc.csv.gz $EVD_PROC
```

8.10 Generate *(since EVL 1.3)*

According to data definition (evd file) generates records to stdout or output flow or file. EVD is EVL data definition file, for details see evl-evd(5).

When no `<config_file>` is specified:

`Number data types`
: values from the whole range of given data type are randomly generated

`Date, timestamp`
: values between 1970-01-01 and 2199-12-31 are randomly generated

`String`
: random characters [a-zA-Z0-9] of the length between 0 and 10 are generated

`Vector`
: random number of elements between 0 and 10 are generated

When `<config_file>` in JSON format is specified:

`Number data types`
: range, values, probability of nulls

`Date, timestamp`
: range, values, probability of nulls

`String`
: range, values, min-length, max-length, probability of nulls

`Vector`
: range, values, min-elements, max-elements, probability of nulls

When both, probability of nulls and values with null is specified, then only probability is taken. When range(s) and values overlaps, then it has no effect on the probability, all values has the same probability of being generated. See examples of JSON below for details.

Synopsis

```
Generate
  <f_out> (<evd>|-d <inline_evd>) [<config_file>]
  [-n|--records <num>] [-y|--text-output]

evl generate
  (<evd>|-d <inline_evd>) [<config_file>]
  [-n|--records <num>] [-y|--text-output]
  [-v|--verbose]

evl generate
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
either this option or the file <evd> must be presented. Example: -d ’id int, user_id string enc=iso-8859-1’

`-n, --records=<num>`
generate <num> number of records instead of the default one

`-y, --text-output`
write the output as text, not binary

Standard options:

`--help`
print this help and exit

`--usage`
print short usage information and exit

`-v, --verbose`
print to stderr info/debug messages of the component

`--version`
print version and exit

Examples

1. Print to stdout one random uchar:

```
evl generate -d 'value uchar' -y
```

2. Example of config JSON file:

```
{
  "int_field": {
    "values": [100, 200, 500],
    "range": { "min": 0, "max": 10 },
    "range": { "min": 50, "max": 60 },
    "null": 0.1
  },
  "float_field": {
    "range": { "min": 0, "max": 100 }
  },
  "date_field": {
    "values": [ null, "2018-03-07", "2018-03-08" ]
  },
  "struct_field.string_field1": {
    "min-length": 10,
    "max-length": 20
  },
  "struct_field.string_field2": {
    "values": ["abc", "def", "ghi", "jkl"]
  },
  "struct_field.decimal_field": {
    "range": { "min": "0.00", "max": "100.00" }
  },
  "vector_field": {
    "min-elements": 2,
    "max-elements": 5
  },
  "vector_field[]": {
    "range": { "min": "2018-03-07 05:00:00", "max": "2018-03-07 14:00:00" }
  }
}
```

where corresponding evd is:

```
int_field        int           sep="|"  null=""
float_field      float         sep="|"
date_field       date          sep="|"  null=""
struct_field     struct        sep="|"
  string_field1  string        sep=";"
  string_field2  string        sep=";"
  decimal_field  decimal(5.2)  sep=";"
vector_field     vector        sep="\n"
  timestamp                    sep=","
```

For the '`int_field`' it will generate randomly values 0,1,...,10,50,...,60,100,200,500, but in 10% cases there will be '`NULL`' values generated.

8.11 Head *(since EVL 1.1)*

Command prints to output first `<num>` records of input. Without option -n prints first 10 records.

`Head`

is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl head`

is intended for standalone usage, i.e. to be invoked from command line.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Head
  <f_in> <f_out> [<evd>|-d <inline_evd>] [-n [-]<num>] [-s|--skip-parse]
  [--validate] [--skip-bom]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]

evl head
  [<evd>|-d <inline_evd>] [-n [-]<num>] [-s|--skip-parse]
  [--validate] [--skip-bom]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]
  [-v|--verbose]

evl head
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-n, --records=[-]<num>`
: output first <num> records instead of the default first 10; or use -n -<num> to output all records except last last <num>

`-s, --skip-parse`
: this option has no effect with '–records <num>' (i.e. the case first <num> records are read and the rest is ignored). But with '–records -<NUM>' it does not parse all fields, but 'jump' over record separator, i.e. the separator of the last field. Be careful with this option, it is particularly good for 'csv' files, when you want to skip some weird formatted footer for example, but might be a wrong solution when some fields are separated by the same character as the last one.

`--skip-bom`
: skip utf-8 BOM (Byte order mark) from the beginning of input, i.e. EF BB BF. Windows usually add it to files in UTF8 encoding

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`--text-input-dos-eol`
: suppose the input as text with CRLF as end of line

`--text-input-mac-eol`
: suppose the input as text with CR as end of line

`-y, --text-output`
: write the output as text, not binary

`--text-output-dos-eol`
: produce the output as text with CRLF as end of line

`--text-output-mac-eol`
: produce the output as text with CR as end of line

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. print to stdout only first 10 records:

   ```
   evl head example.evd -xy <in.txt
   ```

2. read the binary input and omit last 3 records without parsing them (i.e. they no need to have the data structure defined by evd):

   ```
   cat input.bin | evl head -sy -n-3 \
                     -d 'id int sep=",", updated date sep="\n"' \
                       > output.txt
   ```

8.12 Lookup *(since EVL 2.0)*

Prepare lookup from sorted input, which can be used after Wait command till '`Lookup remove`'. Input must be sorted by the `<key>`.

`Lookup [remove]`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl lookup [remove]`
: is intended for standalone usage, i.e. to be invoked from command line.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Lookup
  <f_in> <lookup_name> (<evd>|-d <inline_evd>) -k <key> [-x|--text-input]

Lookup remove
  <lookup_name>

evl lookup
  <lookup_name> (<evd>|-d <inline_evd>) -k <key> [-x|--text-input]
  [-v|--verbose]

evl lookup remove
  <lookup_name>

evl lookup
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-k, --key=<key>`
: key for looking up records

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To prepare lookup at the beginning of the job:

```
Read   dimension.csv DIM     evd/dim.evd  --text-input
Sort   DIM           DIM_SRT evd/dim.evd  --key="id"
Lookup DIM_SRT       dim_lkp evd/dim.evd  --key="id"
```

8.13 Merge *(since EVL 1.2)*

Merge sorted flows or files into one (sorted) output. In the case of only one input flow or file, it simply writes this file to output flow or file.

To merge based on all of the fields, use an empty `<key>`.

`Merge`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl merge`
: is intended for standalone usage, i.e. to be invoked from command line. When `<file>` is '-', then read from stdin.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Merge
  <f_in>... <f_out> [<evd>|-d <inline_evd>] -k|--key <key>
  [-c|--check-sort] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]

evl merge
  [<file>...]  [<evd>|-d <inline_evd>] -k|--key <key>
  [-c|--check-sort] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl merge
  ( --help | --usage | --version )
```

Options

`-c, --check-sort`
: check if the input is really sorted according to specified key

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd_out> must be presented. Example: -d 'some_id long sep="|", some_value string sep="\n"'

`-i, --ignore-case`
: be case insensitive for key fields

`-k, --key=<key>`
: group by this key, where <key> is comma separated list of fields with type (either DESC or ASC, default type is ASC). When the `<key>` is empty, it sorts based on the whole record.

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

`evl merge example.evd -k 'input_id' -y input1.bin input2.bin input3.bin`
: merge three (sorted) binary files, the output is in text and sorted by 'input_id'

8.14 Partition *(since EVL 1.2)*

Read input flow or file and according to '`--key`' or '`--round-robin`' logic send to several number of output flows or files. The number of partitions depends on the '`EVL_PARTITIONS`' environment variable and also on the EVL version/edition.

`Partition`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl partition`
: is intended for standalone usage, i.e. to be invoked from command line.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Partition
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  (--key=<key> | --round-robin)
  [--validate] [-x|--text-input] [-y|--text-output]

evl partition
  <file_in> <file_out> (<evd>|-d <inline_evd>)
  (--key=<key> | --round-robin)
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl partition
  ( --help | --usage | --version | --max-partitions )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd_out> must be presented

`-k, --key=<key>`
: key according to which to distribute data

`-m, --max-partitions`
: return the number of maximal possible partitions

`-r, --round-robin`
: split by round-robin, i.e. simply one record after another to one output flow/file after another

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To partition flow in the EVL job:

```
    Read     s3://my_bucket/cust.csv CUST $EVD_CUST
```

```
Partition   CUST      CUST_P  $EVD_CUST --round-robin
Map         CUST_P    PROC_M  $EVD_CUST $EVD_PROC $EVM_PROC
Departition PROC_M    PROC_G  $EVD_PROC --round-robin
Write       PROC_G    sftp:///some/path/proc.csv.gz $EVD_PROC
```

8.15 Sort *(since EVL 1.0)*

Command takes records from stdin or `<f_in>`, sort them via `<key>` and write them to stdout or `<f_out>`. With the '`-u`' option it deduplicates the data. At the moment it uses only traditional sort order (i.e. like LC_ALL=C), not national.

To sort based on all of the fields, use an empty `<key>`.

`Sort`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl sort`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Sort
  <f_in> <f_out> (<evd>|-d <inline_evd) -k <key>
  [-u <unique-key> [-t|--keep-first] [--reject=<file>]]
  [-c|--check-sort] [-f|--file-storage] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]

evl sort
  (<evd>|-d <inline_evd) -k <key>
  [-u <unique-key> [-t|--keep-first] [--reject=<file>]]
  [-c|--check-sort] [-f|--file-storage] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl sort
  ( --help | --usage | --version )
```

Options

`-c, --check-sort`
: only check if the input is sorted and fail if not

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-f, --file-storage`
: store temporary files on disk instead of using memory

`-i, --ignore-case`
: ignore case sensitivity for key fields

`-k, --key=<key>`
: sort via a key, where `<key>` is comma separated list of fields with type (default type is ASC). When the `<key>` is empty, it sorts based on the whole record. Example: –key='id,user_id DESC,modify_dt ASC'

`-r, --reject=<reject_file>`
: being used with option -u it catch duplicated records into <reject_file>

`-t, --keep-first`
: when deduplicate by –unique-key, keep the first record from the group

`-u, --unique-key=<unique_key>`
: deduplicate the output via <unique_key>; take only the last value unless –keep-first is specified. Duplicated records are catched by -r option. Example: -u 'id,user_id'

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Sort via the whole record (i.e. according to all fields) the text input and write into text output file:

```
evl sort example.evd -k '' -xy <in.txt >out.txt
```

2. Deduplicate the binary input (for example from another EVL component) by keeping the first record in each group with the same id (with the lowest updated date) and write the result into output.csv and duplicates into duplicates.csv:

```
cat input.bin | \
evl sort -ty -k'd,updated' -u'id' \
  -d'id int sep=",", updated date sep="\n"' -r duplicates.csv >output.csv
```

3. Check sort (being case insensitive) of input text file input.txt and write into file output.bin in binary (i.e. not as text):

```
evl sort -cix -k'name' -d'name string sep="|", personal_id int sep="\n"' \
  <input.txt >output.bin
```

8.16 Sortgroup *(since EVL 2.0)*

By having sorted input by `<group_key>`, sort within groups defined by such `<group_key>` according to `<key>`. So output is sorted by `<group_key>`,<key>. At the moment it uses only traditional sort order (i.e. like LC_ALL=C), not national.

`Sortgroup`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl sortgroup`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Sortgroup
  <f_in> <f_out> (<evd>|-d <inline_evd)
  -g|--group-key=<group_key>
  -k|--key=<key>
  [-c|--check-sort] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]

evl sortgroup
  (<evd>|-d <inline_evd)
  -g|--group-key=<group_key>
  -k|--key=<key>
  [-c|--check-sort] [-i|--ignore-case]
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl sortgroup
  ( --help | --usage | --version )
```

Options

`-c, --check-sort`
: check if the input is really sorted by '`;<group_key>`'

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, user_id string'`'

`-g, --group-key=<group_key>`
: input is sorted via this key, where `<group_key>` is comma separated list of fields with type (default type is ASC). Example: '`-k 'id,user_id DESC'`'

`-i, --ignore-case`
: ignore case sensitivity for key fields

`-k, --key=<key>`
: sort via this key within each group of records with same `<group_key>`. `<key>` is comma separated list of fields with type (default type is ASC). Example: '`-k 'modify_dt ASC'`'

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Suppose having a dataset already sorted by field '`customer`'. TBA

8.17 Tail *(since EVL 1.1)*

Command prints to output last `<num>` records of input. Without option '`-n`' prints last 10 records.

`Tail`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl tail`
: is intended for standalone usage, i.e. to be invoked from command line.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Tail
  <f_in> <f_out> [<evd>|-d <inline_evd>] [-n [+]<num>] [-s|--skip-parse]
  [--validate] [--skip-bom]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]


evl tail
  [<evd>|-d <inline_evd>] [-n [+]<num>] [-s|--skip-parse]
  [--validate] [--skip-bom]
  [ -x|--text-input | --text-input-dos-eol | --text-input-mac-eol ]
  [ -y|--text-output | --text-output-dos-eol | --text-output-mac-eol ]
  [-v|--verbose]
```

```
evl tail
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-n, --records=[+]<num>`
: output the last <num> records instead of the default last 10; or use -n +<num> to output starting with record <num>

`-s, --skip-parse`
: with this option it does not parse all fields, but 'jump' over record separator, i.e. the separator of the last field. Be careful with this option, it is particularly good for 'csv' files, when you want to skip some weird formatted header for example, but might be a wrong solution when some fields are separated by the same character as the last one.

`--skip-bom`
: skip utf-8 BOM (Byte order mark) from the beginning of input, i.e. EF BB BF. Windows usually add it to files in UTF8 encoding

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`--text-input-dos-eol`
: suppose the input as text with CRLF as end of line

`--text-input-mac-eol`
: suppose the input as text with CR as end of line

`-y, --text-output`
: write the output as text, not binary

`--text-output-dos-eol`
: produce the output as text with CRLF as end of line

`--text-output-mac-eol`
: produce the output as text with CR as end of line

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Print to stdout only last 10 records:
   ```
   evl tail example.evd -xy <in.txt
   ```
2. Read the binary input and skip first 2 records without parsing them (i.e. they no need to have the data structure defined by evd):
   ```
   cat input.bin | evl tail -sy -n+3 \
                    -d'id int sep=",", updated date sep="\n"'
                       > output.txt
   ```

8.18 Tee *(since EVL 1.0)*

Replicate `<f_in>` to multiple `<f_out>`

Tee

is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

There is no standalone component version as one can use standard UNIX command 'tee'.

Synopsis

```
Tee
  <f_in> <f_out>...

evl tee
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`

print this help and exit

`--usage`

print short usage information and exit

`--version`

print version and exit

Examples

Replicate to output flows (or files) A,B,C,D,E,F:

```
Tee IN_FLOW A B C D E F
```

8.19 Trash *(since EVL 1.0)*

Send `<f_in>` into /dev/null. Try to avoid using it in production environment as redirecting to /dev/null also costs the resources.

`Trash`

is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name, both can be partitioned.

There is no standalone version of this component as you can always use `>`/dev/null.

EVS is EVL job structure definition file, for details see evl-evs(5).

Synopsis

```
Trash
  <f_in>...

evl trash
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

8.20 Uniq *(since EVL 2.1)*

Read stdin or `<f_in>` and write to stdout or `<f_out>` last record in the group specified by the `<key>`. The input must be sorted according to this key.

`Uniq`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl uniq`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Uniq
  <f_in> <f_out> (<evd>|-d <inline_evd>) -k <key> [-c|--check-sort]
  [-i|--ignore-case] [--reject=<file>] [-t|--keep-first]
  [--validate] [-x|--text-input] [-y|--text-output]

evl uniq
  [<evd>] -k <key> [-c|--check-sort]
  [-i|--ignore-case] [--reject=<file>] [-t|--keep-first]
  [--validate] [-x|--text-input] [-y|--text-output]
  [-v|--verbose]

evl uniq
  ( --help | --usage | --version )
```

Options

`-c, --check-sort`
: check if the input is sorted and fail if not

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-i, --ignore-case`
: ignore case sensitivity for key fields

`-k, --key=<key>`
: deduplicate via a key, where <key> is comma separated list of fields with type (default type is ASC). Example: -k 'id,user_id DESC,modify_dt ASC'

`-r, --reject=<reject_file>`
: being used with option -u it catch duplicated records into <reject_file>

`-t, --keep-first`
: keep the first record of the group instead of the last one

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Uniq via the all fields and write into text output file:

```
evl uniq example.evd -k'' -xy < in.txt > out.txt
```

2. Deduplicate the binary input (for example from another EVL component) by keeping the first record in each group with the same id (with the lowest updated date) and write the result into output.csv and duplicates into duplicates.csv:

```
cat input.bin | evl uniq -ty -k'id,updated' -u'id' \
     -d'id int sep=",", updated date sep="\n"' \
     -r duplicates.csv > output.csv
```

3. Check uniq (being case insensitive) of input text file input.txt and write into file output.bin in binary (i.e. not as text):

```
evl uniq -cix --key="name" \
        -d 'name string sep="|", personal_id int sep="\n"' \
        < input.txt > output.bin
```

8.21 Validate *(since EVL 1.1)*

Fail in case invalid data type appear unless '`--limit`' option is specified.

`Validate`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl validate`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD and EVS are definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Validate
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  [-l|--limit <num>] [--text-output]

evl validate
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  [-l|--limit <num>] [--text-output]
  [-v|--verbose]

evl validate
  ( --help | --usage | --version )
```

Options

`-l, --limit=<num>`
: fail after reaching `<num>` number of invalid records. If `<num>` is '`0`', then never fails. Default value is '`1`', i.e. fail immediatelly after first invalid record.

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

8.22 Watcher *(since EVL 1.2)*

This component writes records passing through the `<flow>` into `<file>` in text format.

Works only when variable '`EVL_WATCHER`' is set to '`1`', otherwise does nothing. One can use it for debugging data in '`DEV`' or '`TEST`' environment, but it would be switched off in '`PROD`'.

If not full path to the `<file>` is specified, it writes into directory defined by '`EVL_WATCHER_DIR`' environment variable, which is by default '`watcher`' subfolder of current project.

EVD is EVL data definition file, for details see evl-evd(5).

Synopsis

```
Watcher
  <flow> <file> (<evd>|-d <inline_evd>) [-x|--text-input]

evl watcher
  ( --help | --usage | --version )
```

Options

`-d, --output-definition=<inline_evd>`
: either this option or the file `<evd_out>` must be presented. Example: '`-d 'user_sum long'`'

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. In EVL job ('`evs`' file):

```
Sort     FLOW_01 FLOW_02 some.evd --key='id'
Watcher  FLOW_02 sorted.csv some.evd
```

9 Mapping Components

These components perform a mapping specified by an `evm` file.

- Section 9.1 [Aggreg], page 80, – do aggregations on the groups of records specified by a key,
- Section 9.2 [Join], page 82, – join two inputs by a key,
- Section 9.3 [Map], page 84, – generic mapping component, can be used also for pivoting for example.

9.1 Aggreg *(since EVL 1.0)*

Applies aggregation mapping on each group of records based on the `<key>`.

`Aggreg`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl aggreg`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD, EVM and EVS are EVL definition files, for details see evl-evd(5), evl-evm(5) and evl-evs(5).

Synopsis

```
Aggreg
  <f_in> <f_out> (<evd_in>|-D <inline_evd>) (<evd_out>|-d <inline_evd>)
  <evm> --key=<key>
  [-c|--check-sort] [-i|--ignore-case] [-x|--text-input] [-y|--text-output]
  [-o <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
  [--reject <f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]

evl aggreg
  (<evd_in> | -D <inline_evd>) (<evd_out>|-d <inline_evd>)
  <evm> --key=<key>
  [-c|--check-sort] [-i|--ignore-case] [-x|--text-input] [-y|--text-output]
  [-o|--output <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
  [-r|--reject <f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]
  [-v|--verbose]

evl aggreg
  ( --help | --usage | --version )
```

Options

`-c, --check-sort`
: check if the input is really sorted according to specified key

`-D, --input-definition=<inline_evd>`
: either this option or the file `<evd_in>` must be presented. Example: '`-D 'id int, user_id string'`'

`-d, --output-definition=<inline_evd>`
: either this option or the file `<evd_out>` must be presented. Example: '`-d 'user_sum long'`'

`-i, --ignore-case`
: be case insensitive for key fields

`-k, --key=<key>`
: group by this key, where `<key>` is comma separated list of fields with type (either DESC or ASC, default type is ASC). Example: '`--key='id,user_id DESC'`'

`-o, --output=<f_out>`
: when output() function is used in the mapping, out structure is forwarded into `<f_out>`

`--output<n>=<f_out>`
: when function '`output(<n>)`' is used in mapping, where `<n>` is an integer from 4 to 16, out structure is forwarded into `<f_out>`

`--outputs=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for output(N) functions in mapping. Example: for '`--outputs=OUTFILE`', '`${OUTFILE[120]}`' is the filename used for '`output(120)`'

`-r, --reject=<f_out>`
: when reject() function is used in the mapping, input record is rejected into <f_out>

`--reject<n>=<f_out>`
: when function reject(<n>) is used in mapping, where `<n>` is an integer from 4 to 16, input record is rejected into <f_out>

`--rejects=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for reject(N) functions in mapping. Example: for '`--rejects=REJECTS`', '`${REJECTS[1000]}`' is the filename used for '`reject(1000)`'

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To print to stdout average of amount values:

   ```
   evl aggreg -D 'amount int' -d 'avg int' average.evm -k '' -xy <in.txt
   ```

   File '`average.evm`' might look like this:

   ```
   VARIABLES:
   static int count;
   ```

```
static long sum;

INITIALIZE:
count=0;
sum=0;

COMPUTE:
sum += *in->amount;
count++;

FINALIZE:
*out->avg = sum/count;
```

9.2 Join *(since EVL 1.0)*

Join `<f_left>` and `<f_right>` according to `<key>` and write to `<f_out>` or stdout. Inputs must be sorted by the `<key>`.

Synopsis

```
Join
  <f_left> <f_right> <f_out>
  (<evd_left> | -L <inline_evd>) (<evd_right> | -R <inline_evd>)
  (<evd_out> | -d <inline_evd>) <evm> (-k <key> | -l <key> -r <key> )
  (-t|--type (left|right|inner|outer|cross))
  [-c|--check-sort] [-i|--ignore-case] [-y|--text-output]
  [ [-x|--text-input] | [-a|--left-text-input] [-b|--right-text-input] ]
  [-o <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
  [--reject=<f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]

evl join
  <f_left> <f_right>
  (<evd_left> | -L <inline_evd>) (<evd_right> | -R <inline_evd>)
  (<evd_out> | -d <inline_evd>) <evm> (-k <key> | -l <key> -r <key> )
  (-t|--type (left|right|inner|outer|cross))
  [-c|--check-sort] [-i|--ignore-case] [-y|--text-output]
  [ [-x|--text-input] | [-a|--left-text-input] [-b|--right-text-input] ]
  [-o <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
  [--reject=<f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]
  [-v|--verbose]

evl join
  ( --help | --usage | --version )
```

Options

`-L, --left-definition=<inline_evd>`
    either this option or the file `<evd_left>` must be presented

`-R, --right-definition=<inline_evd>`
    either this option or the file `<evd_right>` must be presented

`-c, --check-sort`
    check if the input is really sorted according to specified key

`-d, --output-definition=<inline_evd>`
: either this option or the file `<evd_out>` must be presented. Example: '`-d 'user_sum long'`'

`-i, --ignore-case`
: be case insensitive for key fields

`-k, --key=<key>`
: join by this key, where <key> is comma separated list of fields with sort type (either DESC or ASC, default type is ASC). This is the shortcut for having the same lists of key fields for '`--key-left`' and '`--key-right`'. Example: '`--key='id,user_id DESC'`'

`-l, --key-left=<key>`
: comma separated list of left fields to join according to Example: '`--key-left='id,name'`'

`-r, --key-right=<key>`
: comma separated list of right fields to join according to Example: '`--key-right='user_id,surname'`'

`-t, --type=<type>`
: mandatory option specifying the join type, possible values for `<type>` are: '`left`', '`right`', '`inner`', '`outer`', '`cross`'

`--unmatched-left=<f_out>`
: when '`unmatched_left()`' function is used in the mapping, out structure is forwarded into `<f_out>`

`--unmatched-right=<f_out>`
: when '`unmatched_right()`' function is used in the mapping, out structure is forwarded into `<f_out>`

`-o, --output=<f_out>`
: when '`output()`' function is used in the mapping, out structure is forwarded into `<f_out>`

`--output<n>=<f_out>`
: when function '`output(<n>)`' is used in mapping, where `<n>` is an integer from 7 to 16, out structure is forwarded into `<f_out>`

`--outputs=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for '`output(N)`' functions in mapping. Example: for '`--outputs=OUTFILE`', '`${OUTFILE[120]}`' is the filename used for '`output(120)`'

`--reject=<f_out>`
: when '`reject()`' function is used in the mapping, input record is rejected into `<f_out>`

`--reject<n>=<f_out>`
: when function '`reject(<n>)`' is used in mapping, where `<n>` is an integer from 7 to 16, input record is rejected into `<f_out>`

`--rejects=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for '`reject(N)`' functions in mapping. Example: for '`--rejects=REJECTS`', '`${REJECTS[1000]}`' is the filename used for '`reject(1000)`'.

`-a, --left-text-input`
: suppose the left input as text, not binary

`-b, --right-text-input`
: suppose the right input as text, not binary

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

9.3 Map *(since EVL 1.0)*

Map input columns to output ones.

`Map`
: is to be used in EVS job structure definition file. `<f_in>` and `<f_out>` are either input and output file or flow name.

`evl map`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input and writing to standard output.

EVD, EVM and EVS are EVL definition files, for details see evl-evd(5), evl-evm(5) and evl-evs(5).

Synopsis

```
     Map
       <f_in> <f_out> (<evd_in>|-D <inline_evd>) (<evd_out>|-d <inline_evd>) <evm>
       [-x|--text-input] [-y|--text-output]
       [-o <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
       [--reject <f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]

     evl map
       (<evd_in>|-D <inline_evd>) (<evd_out>|-d <inline_evd>) <evm>
       [-x|--text-input] [-y|--text-output]
       [-o <f_out>] [--output<n>=<f_out>]... [--outputs=<varname>]
       [--reject=<f_out>] [--reject<n>=<f_out>]... [--rejects=<varname>]
       [-v|--verbose]

     evl map
       ( --help | --usage | --version )
```

Options

`-D, --input-definition=<inline_evd>`
: either this option or the file `<evd_in>` must be presented. Example: '`-D 'id int, user_id string'`'

`-d, --output-definition=<inline_evd>`
: either this option or the file `<evd_out>` must be presented. Example: '`-d 'user_sum long'`'

`-o, --output=<f_out>`
: when '`output()`' function is used in the mapping, out structure is forwarded into `<f_out>`

`--output<n>=<f_out>`
: when function '`output(<n>)`' is used in mapping, where `<n>` is an integer from 4 to 16, out structure is forwarded into `<f_out>`

`--outputs=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for '`output(N)`' functions in mapping. Example: for '`--outputs=OUTFILE`', '`${OUTFILE[120]}`' is the filename used for '`output(120)`'

`--reject=<f_out>`
: when '`reject()`' function is used in the mapping, input record is rejected into `<f_out>`

`--reject<n>=<f_out>`
: when function '`reject(<n>)`' is used in mapping, where `<n>` is an integer from 4 to 16, input record is rejected into `<f_out>`

`--rejects=<varname>`
: specifies an array '`${<varname>[@]}`' which contains filenames to be used for '`reject(N)`' functions in mapping. Example: for '`--rejects=REJECTS`', '`${REJECTS[1000]}`' is the filename used for '`reject(1000)`'

`-x, --text-input`
: suppose the input as text, not binary

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10 Read Components

There is a generic '`Read`' component (see Section 10.1 [Read], page 86), which read source file(s) and parse them based on file suffix from location based on URI Scheme. It can also read a table based on URI.

But for example in the case you need to read and parse particular file format from an input flow, there are also specialized components which parse such format:

Reading various file formats

- Section 10.2 [Readevd], page 91, – parse EVD,
- Section 10.3 [Readjson], page 93, – parse JSON,
- Section 10.7 [Readparquet], page 98, – read Parquet, a columnar file format,
- Section 10.9 [Readqvd], page 101, – read QVD, Qlik's file format,
- Section 10.12 [Readxls], page 105, – parse old-style MS Excel,
- Section 10.13 [Readxlsx], page 106, – read MS Excel,
- Section 10.14 [Readxml], page 107, – parse XML.

And in the case you need to use some DBMS specific options to read a table, there are also DB specific read components:

Reading tables and streams

- Section 10.4 [Readkafka], page 94, – read Kafka topic,
- Section 10.5 [Readmysql], page 95, – read MySQL/MariaDB table,
- Section 10.6 [Readora], page 96, – read Oracle table,
- Section 10.8 [Readpg], page 99, – read PostgreSQL table,
- Section 10.10 [Readsqlite], page 102, – read SQLIte table,
- Section 10.11 [Readtd], page 104, – read Teradata table.

10.1 Read *(since EVL 1.0)*

Read `<source>`(s) (file mask can be specified) and sends it to output `<f_out>`. Multiple `<source>`s are concatenated.

It automatically parses various file formats: '`Avro`', '`json`', '`Parquet`', '`QVD`', '`xls`', '`xlsx`' and '`xml`', just based on file suffix.

Also when compression suffix is recognized, like '`gz`', '`tar`', '`bz2`', '`zip`', '`Z`', data are decompressed automatically.

In general the `<source>` is of the form

[scheme:][//[user@@]host[:port]]/path/basename[.format][.compression]
[scheme:][//[user@@]host[:port]/]database?(table=[schema.]<table>|query=<query>)

When `<source>` starts with '`file:`', '`sftp:`', '`hdfs:`', '`s3:`', '`gs:`' or '`smb:`' it uses appropriate utility to get data from such location. If no URI Scheme is presented, it reads from local file system.

When `<source>` starts with '`mysql:`', '`mssql`', '`postgres:`', '`oracle:`', '`sqlite:`' or '`teradata:`' it uses appropriate utility to get data from such database.

Besides below mentioned options, which changes file suffix behaviour, one can use generic '`--cmd=<cmd>`' option, which calls '`echo <source>... | xargs <cmd>`' to obtain the input for this component. `<cmd>` can be also a pipeline (that is the reason for xargs). See examples below for inspiration.

`Read`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl read`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD is EVL data definition file and EVS defines EVL job structure, for details see evl-evd(5) and evl-evs(5).

URI Scheme for file:

Based on the URI Scheme in the `<source>`, it calls appropriate utility to get files or tables.

`no scheme, 'file:',`
: suppose local filesystem

`'gdrive:'`
: calls '`gdrive`' utility

`'gs:'`
: calls Google's '`gsutil`' utility

`'hdfs:'`
: calls '`hdfs dfs`' utility

`'s3:'`
: calls AWS's '`aws s3`' utility

`'sftp:'`
: calls '`ssh`' utility

`'smb:'`
: calls '`smbclient`' utility

URI Scheme for table:

`'mysql:'`
: calls Readmysql component to read MySQL/MariaDB table

`'mssql:'`
: calls Readmssql component to read MySQL/MariaDB table

`'postgres:'`
: calls Readpg component to read PostgreSQL table

`'oracle:'`
: calls Readora component to read Oracle table

`'sqlite:'`
: calls Readsqlite component to read SQLite table

`'teradata:'`
: calls Readtd component to read Teradata table

Compression:

Compressed file suffix behaviour (applied by following the order):

‘*.tgz’, ‘*.tar.gz’
: calls ‘tar -zxO’

‘*.tar.Z’
: calls ‘tar -ZxO’

‘*.tar.bz2’
: calls ‘tar -jxO’

‘*.tar’
: calls ‘tar -xO’

‘*.gz’, ‘*.GZ’, ‘*.Z’, ‘*.zip’, ‘*.bz2’
: calls ‘gunzip -c’

‘*.zip’, ‘*.ZIP’
: calls ‘unzip -p’

File Type:

Read component behaves according to the <source> suffix.

Specific file formats suffix behaviour:

‘*.avro’, ‘*.AVRO’
: calls ‘evl readavro’

‘*.csv’, ‘*.CSV’, ‘*.txt’, ‘*.TXT’
: read file(s) with ‘--text-input’ option, other than standard Unix end-of-line character (‘\n’) can be specified by option ‘--dos-eol’ or ‘--mac-eol’

‘*.json’, ‘*.JSON’
: calls ‘evl readjson’

‘*.parquet’, ‘*.parq’, ‘*.PARQUET’, ‘*.PARQ’
: calls ‘evl readparquet’

‘*.qvd’, ‘*.QVD’
: calls ‘evl readqvd’

‘*.xls’, ‘*.XLS’
: calls ‘evl readxls’

‘*.xlsx’, ‘*.XLSX’
: calls ‘evl readxlsx’

‘*.xml’, ‘*.XML’
: calls ‘evl readxml’

Synopsis

```
Read
  <source>... <f_out> (<evd>|-d <inline_evd>)
  [--footer=<n>] [--header=<n>] [--cmd=<cmd>]
  [<file_type_options>]
  [--ignore-suffix]  [--allow-missing-file]
  [-y|--text-output [--dos-eol | --mac-eol] ]
  [-w|--where=<condition>] [--filter=<filter>]
  [--validate]
```

```
evl read
  <source>... (<evd>|-d <inline_evd>)
  [--footer=<n>] [--header=<n>] [--cmd=<cmd>]
  [<file_type_options>]
  [--ignore-suffix]  [--allow-missing-file]
  [-y|--text-output [--dos-eol | --mac-eol] ]
  [-w|--where=<condition>] [--filter=<filter>]
  [--validate]
  [-v|--verbose]

evl read
  ( --help | --usage | --version )
```

Options

`--allow-missing-file`
: don't fail if `<source>` doesn't exist, and produce empty output

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d \"user_name string, user_sum int\"`'

`--filter=<filter>`
: when '`--where`' option is used and replacing of SQL syntax is not valid, use `<filter>` when reading file(s)

`-f, --footer=<n>`
: skip last <n> records. When multiple files, skip last `<n>` records in each of them. Command '`evl head -n-<n> --skip-parse`' is used for this job.

`-h, --header=<n>`
: skip first `<n>` records. When multiple files, skip first `<n>` records in each of them. Command '`evl tail -<n>+(N+1) --skip-parse`' is used for this job.

`--cmd=<cmd>`
: bash command <cmd> is used to read the <source>s. In such case recognizing file's suffix is switched off. See examples below for inspiration.

`--ignore-suffix`
: ignore <source>'s suffix, act only based on options.

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-w, --where=<condition>`
: use this where condition instead of reading whole file/table. In case of reading a table it sends the query to the database with this where condition. In case of a file it reads the whole file and apply the evl-filter component right after. For the filter it replaces these SQL logical operators to C++ ones:

- '`AND`' -> '`&&`'
- '`OR`' -> '`||`'
- '`=`' -> '`==`'
- <> -> '`!=`'

so one can use also SQL notation to specify a condition. It also removes quotes around field names and replaces single quotes by double quotes for proper string notion:

- '`\"field_name\"`' -> '`field_name`'
- '`'`' -> '`\"`'

Can be useful to have the same syntax for files and for tables.

`-x, --text-input`
: suppose the input as text, not binary

`--dos-eol`
: suppose the input is text with CRLF as end of line

`--mac-eol`
: suppose the input is text with CR as end of line

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

File type options:

`--avro`
: whatever <source>'s suffix, act as reading '`avro`' file format

`--gz`
: whatever <source>'s suffix, act as reading '`gz`', '`Z`', '`zip`', '`bz2`' compressed file format

`--json`
: whatever <source>'s suffix, act as reading '`json`' file format

`--parquet`
: whatever <source>'s suffix, act as reading '`parquet`' file format

`--qvd`
: whatever <source>'s suffix, act as reading Qlik's '`QVD`' file format

`--xls, --xlsx`
: whatever <source>'s suffix, act as reading MS Excel '`xls`' or '`xlsx`' file format

`--tar`
: whatever <source>'s suffix, act as reading tar file

`--xml`
: whatever <source>'s suffix, act as reading '`xml`' file format

QVD, XLS, XLSX, XML and JSON specific option:

`--match-fields`
: for other than QVD, XLS(X), XML and JSON file is this option ignored.

XML and JSON specific option:

`--all-fields-exist`
: for other then XML and JSON file is this option ignored.

XML specific options:

`--document-tag=<tag>`
: for other then XML file is this option ignored. Check '`man evl readxml`' for details.

`--record-tag=<tag>`
: for other then XML file is this option ignored. Check '`man evl readxml`' for details.

`--vector-element-tag=<tag>`
: for other then XML file is this option ignored. Check '`man evl readxml`' for details.

XLS and XLSX specific options:

`--sheet-index=<n>`
: read `<n>`-th sheet, starting from number 0. '`--sheet-index=0`' is default

`--sheet-name=<name>`
: read sheet with name `<name>`

Examples

Standard examples of standalone usage:

1. Read tar.gz, skip header line and validate data types Write into '`example.csv`' the content of the tarred and gzipped source without the header line and with validated data types:

```
evl read -d 'id int sep=";", value string sep="\n"' \
         -h1 -vxy <example.csv.tar.gz >example.csv
```

2. Gzipped json file:

```
evl read sample.json.gz sample.evd -y >sample.csv
```

As the file has standard file suffixes '`gz`' and '`json`', they are automatically recognized a gunzipped and parsed as JSON.

Standard examples of usage in EVL Job:

3. Gzipped json file. The same as example 2., but to be used in evs file:

```
Read   sample.json.gz SRC sample.evd
Write  SRC     sample.csv sample.evd
```

10.2 Readevd *(since EVL 2.5)*

Read EVD file from stdin and output using this evd structure:

```
parents vector null=""
  string
name string
data_type string
format string null=""
```

```
comment string null=""
null vector null=""
  string
separator string null=""
quote struct null=""
  char string(1)
  optional uchar
options vector
  struct
    tag string
    value string null=""
decimal struct null=""
  precision uchar
  scale uchar
  decimal_separator string(1) null=""
  thousands_separator string null=""
string struct null=""
  length ulong null=""
  locale string null=""
  encoding string null=""
  max_bytes ulong null=""
  max_chars ulong null=""
ustring struct null=""
  length ulong null=""
  locale string null=""
  encoding string null=""
  max_bytes ulong null=""
  max_chars ulong null=""
```

`Readevd`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readevd`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readevd
  <f_in> <f_out> [-y|--text-output]

evl readevd
  [-y|--text-output] [-v|--verbose]

evl readevd
  ( --help | --usage | --version )
```

Options

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.3 Readjson *(since EVL 1.2)*

Parse `<f_in>` into `<evd>`.

In general not all input fields need to exist in the input JSON, but if they are, then the option '`--all-fields-exist`' will speed up the processing.

When the input JSON has not the same order of fields as defined in `<evd>`, then option '`--match-fields`' has to be used.

Usually when reading JSON file written by '`Writejson`', it is good to call '`Readjson`' with option '`-a`', as there are always all fields from `<evd>`.

`Readjson`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readjson`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readjson
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  [-a|--all-fields-exist] [-m|--match-fields] [-y|--text-output]

evl readjson
  (<evd>|-d <inline_evd>)
  [--all-fields-exist] [--match-fields] [-y|--text-output]
  [-v|--verbose]

evl readjson
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'user_sum long'

`-a, --all-fields-exist`
: when the input contain all fields (e.g. output of evl-writejson), then using this option increase the performance

`-m, --match-fields`
: when field are not in the same order as used in evd, this option must be used

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

In general not all input fields need to exist in the input JSON, but if they are, then the option "–all-fields-exist" will speed up the processing.

When the input JSON has not the same order of fields as defined in "EVD" file, then option "–match-fields" has to be used.

Usually when reading file written by "EVL" component ‘`Writejson`’, it is good to call "Readjson" with option "-a", as there are always all fields from "EVD".

10.4 Readkafka *(since EVL 1.1)*

Component calls kafka consumer command, specified by ‘`EVL_KAFKA_CONSUMER_COMMAND`’, which is by default ‘`kafka-console-consumer.sh`’. and run it with options:

```
--bootstrap-server "<server>:<port>" --topic "<topic>" ``<kafka_consumer_opts>``
```

and send the output to `<f_out>`.

`Readkafka`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readkafka`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVS is EVL job structure definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readkafka
  <topic> <f_out>
  -s|--bootstrap-server <server:port>
  [<kafka_consumer_opts>]

evl readkafka
  <topic>
  -s|--bootstrap-server <server:port>
  [<kafka_consumer_opts>]
  [-v|--verbose]
```

```
evl readkafka
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.5 Readmysql *(since EVL 2.4)*

Write to stdout or `<f_out>` MariaDB/MySQL `<table>`.

Password is taken from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`'. When such file has not permissions 600 (or 400), it is ignored! For details see '`evl-password`'.

`Readmysql`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readmysql`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readmysql
  [<schema>.]<table> <f_out> (<evd>|-d <inline_evd>)
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-q|--query=<query>] [-u|--username=<mysqluser>]
  [--mysql=<mysql-options>] [-y|--text-output]

evl readmysql
  [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-q|--query=<query>] [-u|--username=<mysqluser>]
  [--mysql=<mysql-options>] [-y|--text-output]
  [-v|--verbose]

evl readmysql
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`-q, --query=<query>`
: Use SQL <query> instead of reading whole table. With this option <table> might be an empty string.

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

'mysql' options:

`-b, --dbname=<database>`
: this option is provided to '`mysql`' command as '`--database=<database>`'

`-h, --host=<hostname>`
: this option is provided to '`mysql`' command

`-p, --port=<port>`
: using other than standard port 3306. This option is provided to '`mysql`' command.

`-u, --username=<mysqluser>`
: if not mentioned, then current system username is used as mysql user. This option is provided to '`mysql`' command as '`--user=<mysqluser>`'.

`--mysql=<mysql-options>`
: other mysql options can be specified here

10.6 Readora *(since EVL 2.0)*

Write to standard output or `<f_out>` Oracle `<table>`.

When `<schema>` is not present, environment variable '`ORADATABASE`' is used.

Password is taken from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`'. When such file has not permissions 600 (or 400), it is ignored! For details see '`evl-password`'.

`Readora`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readora`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

`SQL*Plus Field Separator`
: Reading the table by SQL*Plus uses as field seprator the value of '`$EVL_ORACLE_FIELD_SEPARATOR`', which is by default set to '`\x1f`' (i.e. an Unit Separator), and last field in each record is separated by '`\n`'.

`SQL*Plus script hook`
: Custom options might be added to SQL*Plus script by environment variable '`$EVL_ORACLE_SQLPLUS_HOOK`'.

Synopsis

```
     Readora
       [<schema>.]<table> <f_out> <evd>
       [--query=<query>] [-w|--where=<condition>]
       [ --connect=<connect_identifier> | -b|--dbname=<database> -h|--host=<hostname> [-p|-
       [-u|--username=<oracle_user>] [-y|--text-output]

     evl readora
       [<schema>.]<table> <evd>
       [--query=<query>] [-w|--where=<condition>]
       [ --connect=<connect_identifier> | -b|--dbname=<database> -h|--host=<hostname> [-p|-
       [-u|--username=<oracle_user>] [-y|--text-output]
       [-v|--verbose]

     evl readora
       ( --help | --usage | --version )
```

Options

`--query=<query>`
: use SQL `<query>` instead of reading whole table. With this option `<table>` might be an empty string.

`-w, --where=<condition>`
: use this where condition instead of reading whole table.

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

'sqlplus' options:

`--connect=<connect_identifier>`
: sqlplus will be called in the form:

<username>/<password>@<connect_identifier>

where <connect_identifier> can be in the form

[<net_service_name> | [//]Host[:Port]/<service_name>]

without this option environment variable 'ORACONN' (if defined) is used as connection identifier for sqlplus

`-b, --dbname=<database>`
: either this or environment variable 'ORADATABASE' should be provided, If also 'ORADATABASE' environment variable is set, this option has preference.

`-h, --host=<hostname>`
: either this or environment variable 'ORAHOST' should be provided when connecting to other host than localhost. If also 'ORAHOST' variable is set, this option has preference.

`-p, --port=<port>`
: either this or environment variable 'ORAPORT' should be provided when using other than standard port '1521'.

`-u, --username=<oracle_user>`
: without this option environment variable 'ORAUSER' is used as user for sqlplus

10.7 Readparquet *(since EVL 2.0)*

Write to stdout or `<f_out>` Parquet files from `<parquet>` directory.

`Readparquet`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readparquet`
: is intended for standalone usage, i.e. to be invoked from command line and writing records into standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readparquet
  <parquet> <f_out> (<evd>|-d <inline_evd>) [-y|--text-output]

evl readparquet
  <parquet> (<evd>|-d <inline_evd>) [-y|--text-output]
  [-v|--verbose]

evl readparquet
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, name string, started timestamp'

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.8 Readpg *(since EVL 2.0)*

Write to standard output or `<f_out>` PostgreSQL `<table>`.

Password is taken:

1. from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`',
2. from file '`$PGPASSFILE`', which is by default '`$HOME/.pgpass`'.

When such file has not permissions 600, it is ignored! For details see '`evl-password`'.

`Readpg`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readpg`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readpg
  [<schema>.]<table> <f_out> (<evd>|-d <inline_evd>)
  [-q|--query=<query> | -w|--where=<condition>]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>] [-y|--text-output]

evl readpg
  [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-q|--query=<query> | -w|--where=<condition>]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>] [-y|--text-output]
  [-v|--verbose]

evl readpg
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or `<evd>` file must be presented. Example: '`-d 'id int, user_id string enc=iso-8859-1'`'

`-q, --query=<query>`
: Use SQL `<query>` instead of reading whole table. With this option `<table>` might be an empty string.

`-w, --where=<condition>`
: use this where condition instead of reading whole table

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

'psql' options:

`-b, --dbname=<database>`
: either this or environment variable '`PGDATABASE`' should be provided, if not, then current system username is used as psql database. If also '`PGDATABASE`' environment variable is set, this option has preference. (This option is provided to '`psql`' command.)

`-h, --host=<hostname>`
: either this or environment variable '`PGHOST`' should be provided when connecting to other host than localhost. If also '`PGHOST`' variable is set, this option has preference. (This option is provided to '`psql`' command.)

`-p, --port=<port>`
: either this or environment variable '`PGPORT`' should be provided when using other than standard port '`5432`'. (This option is provided to '`psql`' command.)

`--psql=<psql_options>`
: all other options to be provides to psql command. See '`man psql`' for details.

`-u, --username=<pguser>`
: either this or environment variable '`PGUSER`' should be provided, if not, then current system username is used as psql user. If variable '`PGUSER`' is set, this option has preference. (This option is provided to '`psql`' command.)

Examples

1. To read a table from default schema (mostly '`public`') in EVL job (i.e. in EVS file) from localhost:5432:

   ```
   export PGUSER=some_pg_user
   export PGDATABASE=my_db
   Readpg my_table MYTABLE evd/mytable.evd
   Map    MYTABLE  ...
   ```

   Password is taken from ~/.pgpass, which has 600 permissions and look like this:

   ```
   localhost:5432:my_db:some_pg_user:H+SCs9;_@D
   ```

10.9 Readqvd *(since EVL 2.3)*

Write to standard output or `<f_out>` the content of the `<file.qvd>`. It parses fields as they are specified in EVD file, unless '`--match-fields`' is specified.

If there are less fields in the EVD file than in QVD, only such fields are returned.

`Readqvd`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readqvd`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readqvd
  <file.qvd> <f_out> (<evd>|-d <inline_evd>)
  [-y|--text-output | -a|--text-output-dos-eol | -b|--text-output-mac-eol]
  [-m|--match-fields]
  [-n|--null-as-string[=<string>]]
  [--filter=<condition>]
  [--first-record=<n>]
  [--guess-uniform-symbol-size]
  [--low-memory]

evl readqvd
  <file.qvd> (<evd>|-d <inline_evd>)
  [-y|--text-output | -a|--text-output-dos-eol | -b|--text-output-mac-eol]
  [-m|--match-fields]
  [-n|--null-as-string[=<string>]]
  [--filter=<condition>]
  [--first-record=<n>]
  [--guess-uniform-symbol-size]
  [--low-memory]
  [-v|--verbose]

evl readqvd
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, name string, started timestamp'`'

`-m, --match-fields`
: match fields between EVD and QVD, otherwise they are taken one by one from input QVD file. If there are less fields in the EVD file than in QVD, only such fields are returned.

`-n, --null-as-string[=<string>]`
: read `<string>` as a NULL value, without `<string>` specified it reads an empty string as NULL

`--filter=<condition>`
: read only records with given `<condition>`.

`--first-record=<n>`
: start to read from the record number `<n>`.

`--guess-uniform-symbol-size`
: might speed up indexing of dictionary, but it could not work in all cases. Use only in special cases when need really good performance.

`--low-memory`
: do not read dictionary into memory. This could save memory consumption, but slows down reading the source file.

`-y, --text-output`
: write the output as text, not binary

`--text-output-dos-eol`
: produce the output as text with CRLF as end of line

`--text-output-mac-eol`
: produce the output as text with CR as end of line

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.10 Readsqlite *(since EVL 2.7)*

Write to stdout or `<f_out>` SQLite `<table>`.

It takes the whole table with columns in order defined by EVD, unless `<query>` and/or `<condition>` is specified.

Path to the database file is taken from environment variable '`$EVL_SQLITE_DATABASE`', unless `<db_file>` is specified.

`Readsqlite`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readsqlite`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readsqlite
  <table> <f_out> (<evd>|-d <inline_evd>)
  [--dbname=<db_file>] [--query=<query>] [-w|--where=<condition>]
  [-y|--text-output]

evl readsqlite
  <table> (<evd>|-d <inline_evd>)
  [--dbname=<db_file>] [--query=<query>] [-w|--where=<condition>]
  [-y|--text-output]
  [-v|--verbose]

evl readsqlite
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, user_id string enc=iso-8859-1'

`--dbname=<db_file>`
: path to the SQLite database file; if this option is not used, database file is taken from environment variable '`$EVL_SQLITE_DATABASE`'.

`--query=<query>`
: Use SQL <query> instead of reading whole table. With this option <table> might be an empty string.

`-w, --where=<condition>`
: use this where condition instead of reading whole table.

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To read a table '`my_table`' in EVL job (i.e. in EVS file) from '`/home/myself/my_db.sqlite`':

   ```
   export EVL_SQLITE_DATABASE="/home/myself/my_db.sqlite"
   Readsqlite my_table MYTABLE evd/mytable.evd
   Map        MYTABLE  ...
   ```

2. Command line usage of sending table ‘my_table’ from ‘/home/myself/my_db.sqlite’ to standard output:

   ```
   export EVL_SQLITE_DATABASE="/home/myself/my_db.sqlite"
   evl readsqlite my_table evd/mytable.evd --text-output
   ```

   or just

   evl readsqlite –dbname="/home/myself/my_db.sqlite" my_table evd/mytable.evd –text-output

10.11 Readtd *(since EVL 1.1)*

Write to stdout or <f_out> Teradata <table>.

`Readtd`
: is to be used in EVS job structure definition file. <f_out> is either output file or flow name.

`evl readtd`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readtd
  <database>.<table> <f_out> (<evd>|-d <inline_evd>) [-y|--text-output]

evl readtd
  <database>.<table> (<evd>|-d <inline_evd>) [-y|--text-output]
  [-v|--verbose]

evl readtd
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d ’id int, user_id string enc=iso-8859-1’

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.12 Readxls *(since EVL 2.2)*

Read XLS sheet and write to `<f_out>`.

Unless '`--sheet-index`' or '`--sheet-name`' is specified, it reads only the first sheet from the file.

It skips the header line, unless option '`--no-header`' or '`--match-fields`' is used.

`Readxls`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readxls`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readxls
  <file> <f_out> (<evd>|-d <inline_evd>)
  [-m|--match-fields | --no-header]
  [--sheet-index=<n> | --sheet-name=<name>]
  [-y|--text-output]

evl readxls
  <file> (<evd>|-d <inline_evd>)
  [-m|--match-fields | --no-header]
  [--sheet-index=<n> | --sheet-name=<name>]
  [-y|--text-output]
  [-v|--verbose]

evl readxls
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, name string, started timestamp'`'

`-m, --match-fields`
: read only fields specified by EVD, based on header. All characters other than '`[a-zA-Z0-9_-]`' are replaced by underscore when matching with EVD field names.

`--no-header`
: suppose there is no header

`--sheet-index=<n>`
: read `<n>`-th sheet, starting from number 0 (i.e. '`--sheet-index=0`' is the default behaviour)

`--sheet-name=<name>`
: read sheet with name `<name>`

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.13 Readxlsx *(since EVL 2.2)*

Read XLSX sheet and write to `<f_out>`.

Unless '`--sheet-index`' or '`--sheet-name`' is specified, it reads only the first sheet from the file.

It skips the header line, unless option '`--no-header`' or '`--match-fields`' is used.

`Readxlsx`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readxlsx`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readxlsx
  <file> <f_out> (<evd>|-d <inline_evd>)
  [-m|--match-fields | --no-header]
  [--sheet-index=<n> | --sheet-name=<name>]
  [-y|--text-output]

evl readxlsx
  <file> (<evd>|-d <inline_evd>)
  [-m|--match-fields | --no-header]
  [--sheet-index=<n> | --sheet-name=<name>]
  [-y|--text-output]
  [-v|--verbose]

evl readxlsx
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, name string, started timestamp'`'

`-m, --match-fields`
: read only fields specified by EVD, based on header. All characters other than ‘`[a-zA-Z0-9_-]`’ are replaced by underscore when matching with EVD field names.

`--no-header`
: suppose there is no header

`--sheet-index=<n>`
: read `<n>`-th sheet, starting from number 0 (i.e. ‘`--sheet-index=0`’ is the default behaviour)

`--sheet-name=<name>`
: read sheet with name `<name>`

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

10.14 Readxml *(since EVL 1.3)*

Parse XML `<f_in>` into `<evd>`.

In general not all input fields need to exist in the input XML, but if they are, then the option ‘`--all-fields-exist`’ will speed up the processing.

When the input XML has not the same order of fields as defined in `<evd>`, then option ‘`--match-fields`’ has to be used.

Usually when reading XML file written by ‘`Writexml`’ it is good to call ‘`Readxml`’ with option ‘`-a`’, as there are always all fields from `<evd>`.

`Readxml`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl readxml`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Readxml
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  [-a|--all-fields-exist]
  [-m|--match-fields]
```

```
    [--document-tag=<tag>]
    [--record-tag=<tag>]
    [--vector-element-tag=<tag>]
    [-y|--text-output]

  evl readxml
    (<evd>|-d <inline_evd>)
    [-a|--all-fields-exist]
    [-m|--match-fields]
    [--document-tag=<tag>]
    [--record-tag=<tag>]
    [--vector-element-tag=<tag>]
    [-y|--text-output]
    [-v|--verbose]

  evl readxml
    ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
either this option or the file <evd> must be presented. Example: -d 'user_sum long'

`-a, --all-fields-exist`
: when the input contain all fields (e.g. output of evl-writexml), then using this option increase the performance

`-m, --match-fields`
: when field are not in the same order as used in evd, this option must be used

`--document-tag=<tag>`
specify a tag name of the main tag, by default it tries to guess it. XML file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  ...
</document>
```

where the tag '`document`' can be of any name.

`--record-tag=<tag>`
specify a tag name of a record, by default it tries to guess it. XML file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <record>
    ...
  </record>
  <record>
    ...
  </record>
  <record>
    ...
  </record>
  ...
```

```
</document>
```

where the tag '`record`' can be of any name, but the same accross the file.

`--vector-element-tag=<tag>`
: the name of the tag for vector elements, e.g. XML file with vector '`someVector`':

```
...
<someVector>
  <elem>1</elem>
  <elem>2</elem>
  <elem>3</elem>
</someVector>
...
```

shoul be read with option '`--vector-element-tag=elem`'.

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

11 Run SQL Components

Beside the ‘Read’ components, which read tables from databases, there are also components which run SQL in database.

Runin tables and streams

- Section 11.1 [Runmysql], page 110, – run SQL or mysql command in MariaDB/MySQL database,
- Section 11.2 [Runora], page 111, – run SQL in Oracle database,
- Section 11.3 [Runpg], page 112, – run SQL or psql command in PostgreSQL database,
- Section 11.4 [Runsqlite], page 113, – run SQL or sqlite command in SQLite database.

11.1 Runmysql *(since EVL 2.4)*

Run SQL or mysql commands from stdin or `<f_in>` and write result into stdout or `<f_out>`. It returns output from ‘`mysql`’ as is, so for quering the table to get formatted EVL output use ‘`Readmysql`’ or ‘`evl readmysql`’.

Password is taken from $HOME/.mysqlpass file, unless other file is specified by ‘`--defaults-extra-file=<file>`’.

`Runmysql`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl runmysql`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Runmysql
  <f_in> <f_out> [--dbname=<database>] [--host=<hostname>]
  [--port=<port>] [--username=<mysqluser>] [--mysql=<mysql_options>]

evl runmysql
  [--dbname=<database>] [--host=<hostname>]
  [--port=<port>] [--username=<mysqluser>] [--mysql=<mysql_options>]
  [-v|--verbose]

evl runmysql
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

'mysql' options:

`--dbname=<database>`
: this option is provided to '`mysql`' command as '`--database=<database>`'

`--host=<hostname>`
: this option is provided to '`mysql`' command

`--port=<port>`
: using other than standard port 3306. This option is provided to '`mysql`' command.

`--username=<mysqluser>`
: if not mentioned, then current system username is used as mysql user. This option is provided to '`mysql`' command as '`--user=<mysqluser>`'.

`--mysql=<mysql_options>`
: all `<mysql_options>` is provided to '`mysql`' utility.

11.2 Runora *(since EVL 2.0)*

Run SQL from stdin or `<f_in>` and write result into stdout or `<f_out>`. It returns output from 'sqlplus' as is, so for quering the table to get formatted EVL output use '`Readora`' or '`evl readora`'.

`Runora`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl runora`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Runora
  <f_in> <f_out>
  [-u|--username=<user>] [-p|--password=<password>]
  [--connect=<connect_identifier>]

evl runora
  [-u|--username=<user>] [-p|--password=<password>]
  [--connect=<connect_identifier>]
  [-v|--verbose]

evl runora
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

'sqlplus' options:

`--connect=<connect_identifier>`
: sqlplus will be called in the form:

  <username>/<password>@<connect_identifier>

  where <connect_identifier> can be in the form

  [<net_service_name> | [//]Host[:Port]/<service_name>]

  without this option environment variable ORACONN (if defined) is used as connection identifier for sqlplus

`-p, --password=<password>`
: without this option environment variable ORAPASS is used as password for sqlplus

`-u, --username=<user>`
: without this option environment variable ORAUSER is used as user for sqlplus

11.3 Runpg *(since EVL 2.0)*

Run SQL or psql commands from stdin or `<f_in>` and write result into stdout or `<f_out>`. It returns output from '`psql`' as is, so for quering the table to get formatted EVL output use '`Readpg`' or '`evl readpg`'.

Password is taken:

1. from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`',
2. from file '`$PGPASSFILE`', which is by default '`$HOME/.pgpass`'.

When such file has not permissions 600, it is ignored! For details see '`evl-password`'.

`Runpg`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl runpg`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Runpg
  <f_in> <f_out>
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>]
```

```
     evl runpg
       [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
       [-u|--username=<pguser>] [--psql=<psql_options>]
       [-v|--verbose]

     evl runpg
       ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

'psql' options:

`-b, --dbname=<database>`
: either this or environment variable '`PGDATABASE`' should be provided, if not, then current system username is used as psql database. If also '`PGDATABASE`' environment variable is set, this option has preference. (This option is provided to 'psql' command.)

`-h, --host=<hostname>`
: either this or environment variable '`PGHOST`' should be provided when connecting to other host than localhost. If also '`PGHOST`' variable is set, this option has preference. (This option is provided to '`psql`' command.)

`-p, --port=<port>`
: either this or environment variable '`PGPORT`' should be provided when using other then standard port '`5432`'. (This option is provided to '`psql`' command.)

`--psql=<psql_options>`
: by default it runs '`psql`' command behind with these options:

```
      ``--no-align --quiet --tuples-only --set=\"ON_ERROR_STOP=1\"``
```

If any other option is needed, specify `<psql_options>`. See '`man psql`' for details.

`-u, --username=<pguser>`
: either this or environment variable '`PGUSER`' should be provided, if not, then current system username is used as psql user. If variable '`PGUSER`' is set, this option has preference. (This option is provided to '`psql`' command.)

11.4 Runsqlite *(since EVL 2.7)*

Run SQL or sqlite3 commands from stdin or `<f_in>` and write result into stdout or `<f_out>`. It returns output from '`sqlite3`' as is, so for quering the table to get formatted EVL output use '`Readsqlite`' or '`evl readsqlite`'.

Path to the database file is taken from environment variable '`$EVL_SQLITE_DATABASE`', unless `<db_file>` is specified.

`Runsqlite`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl runsqlite`
: is intended for standalone usage, i.e. to be invoked from command line and writing records to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Runsqlite
  <f_in> <f_out> [--dbname=<db_file>] [--sqlite=<sqlite_options>]

evl runsqlite
  [--dbname=<db_file>] [--sqlite=<sqlite_options>]
  [-v|--verbose]

evl runsqlite
  ( --help | --usage | --version )
```

Options

`--dbname=<db_file>`
: path to the SQLite database file; if this option is not used, database file is taken from environment variable '`$EVL_SQLITE_DATABASE`'.

`--sqlite=<sqlite_options>`
: all `<sqlite_options>` is provided to '`sqlite`' utility.

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12 Write Components

Beside the generic '`Write`' component, which behaves according to file(s) suffixes, there are also components which produce particular file format or connect to database and write a table.

There are two groups of write components:

Writing various file formats

- Section 12.2 [Writeevd], page 120, – write EVD,
- Section 12.3 [Writejson], page 122, – write JSON,
- Section 12.7 [Writeparquet], page 127, – write Parquet, a columnar file format,
- Section 12.9 [Writeqvd], page 130, – write QVD, Qlik's file format,
- Section 12.10 [Writeqvx], page 131, – write QVX, Qlik's file format,
- Section 12.13 [Writexlsx], page 134, – write MS Excel,
- Section 12.14 [Writexml], page 136, – write XML.

Writing tables and streams

- Section 12.4 [Writekafka], page 123, – write Kafka topic,
- Section 12.5 [Writemysql], page 124, – write MySQL/MariaDB table,
- Section 12.6 [Writeora], page 125, – write Oracle table,
- Section 12.8 [Writepg], page 128, – write PostgreSQL table,
- Section 12.11 [Writesqlite], page 132, – write SQLite table,
- Section 12.12 [Writetd], page 133, – write Teradata table.

12.1 Write *(since EVL 1.0)*

Write `<f_in>` into `<target>` which is a file or table specified in general by

[scheme:][//[user@@]host[:port]]/path/basename[.format][.compression]
[scheme:][//[user@@]host[:port]/]database?(table=[schema.]<table>|query=<query>)

Besides below mentioned options, which changes file suffix behaviour, one can use generic '`--cmd=<cmd>`' option, which calls something like '`| <cmd> > <path>`' at the end. `<cmd>` can be also a pipeline. See examples below for inspiration.

`Write`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl write`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD is EVL data definition file, for details see evl-evd(5).

URI Scheme:

Based on the URI Scheme '`scheme:`', component calls appropriate utilities to write the file to the destination.

`no scheme`, '`file:`',
: suppose local filesystem

'`gdrive:`'
: calls '`gdrive`' utility

‘`gs:`’
: calls ‘`gsutil`’ utility

‘`hdfs:`’
: calls ‘`hadoop fs`’ utility

‘`s3:`’
: calls ‘`aws s3`’ utility

‘`sftp:`’
: calls ‘`ssh`’ utility

‘`smb:`’
: calls ‘`smbclient`’ utility

Based on the URI Scheme ‘`scheme:`’, component switch to appropriate EVL component.

‘`mysql:`’
: calls Writemysql component to write to MySQL/MariaDB table

‘`mssql:`’
: calls Writemssql component to write to MySQL/MariaDB table

‘`oracle:`’
: calls Writeora component to write to Oracle table

‘`postgres:`’
: calls Writepg component to write to PostgreSQL table

‘`sqlite:`’
: calls Writesqlite component to write to SQLite table

‘`teradata:`’
: calls Writetd component to write to Teradata table

Compression:

Compression file suffix behaviour (applied by following the order):

‘`*.bz2`’, ‘`*.BZ2`’
: calls ‘`bzip2 -c`’

‘`*.gz`’, ‘`*.GZ`’
: calls ‘`gzip -c`’

‘`*.zip`’, ‘`*.ZIP`’
: calls ‘`zip`’

File Type:

Write component behaves according to the `<file>` suffix.

Specific file formats suffix behaviour:

‘`*.avro`’, ‘`*.AVRO`’
: calls ‘`evl writeavro`’

‘`*.csv`’, ‘`*.CSV`’, ‘`*.txt`’, ‘`*.TXT`’
: write file with ‘`--text-output`’ option, other than standard Unix end-of-line character (‘`\n`’) can be specified by option ‘`--dos-eol`’ or ‘`--mac-eol`’

‘`*.json`’, ‘`*.JSON`’
: calls ‘`evl writejson`’

'*.parquet', '*.parq', '*.PARQUET', '*.PARQ'
calls 'evl writeparquet'

'*.qvd', '*.QVD'
calls 'evl writeqvd'

'*.qvx', '*.QVX'
calls 'evl writeqvx'

'*.xlsx', '*.XLSX'
calls 'evl writexlsx'

'*.xml', '*.XML'
calls 'evl writexml'

Synopsis

```
Write
  <f_in> <target> (<evd>|-d <inline_evd>)
  [-a|--append]
  [--footer-file=<f_in>] [--header-file=<f_in> | -h|--header]
  [ --avro |
    --json [--omit-null-fields] [--array-output] |
    --parquet |
    --qvd | --qvx
    --xlsx
    --xml [--document-tag=<tag>] [--record-tag=<tag>]
          [--vector-element-tag=<tag>] |
    -y|--text-output [--dos-eol] [--mac-eol]
  ]
  [--gz] [--cmd=<cmd>] [--ignore-suffix]
  [-x|--text-input] [--validate]

evl write
  <target> (<evd>|-d <inline_evd>)
  [-a|--append]
  [--footer-file=<file>] [--header-file=<file> | -h|--header]
  [ --avro |
    --json [--omit-null-fields] [--array-output] |
    --parquet |
    --qvd | --qvx
    --xlsx
    --xml [--document-tag=<tag>] [--record-tag=<tag>]
          [--vector-element-tag=<tag>] |
    -y|--text-output [--dos-eol] [--mac-eol]
  ]
  [--gz] [--cmd=<cmd>] [--ignore-suffix]
  [-x|--text-input] [--validate]
  [-v|--verbose]

evl write
  ( --help | --usage | --version )
```

Options

Common options:

`-a, --append`
: do not overwrite the target file or table, but only append. When used with file formats Avro, Parquet, QVD, QVX or XLSX, warning is displayed and target file is overwritten. For these formats append doesn't make sense or is not possible. So better use this option with care. Rather concatenate the increment with previous version of the file/table and then move over.

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented

`--footer-file=<file>`
: add `<file>` after last written record When used with file formats Parquet, QVD, QVX or XLSX, warning is displayed and no `<file>` is appended. (It doesn't make sense for these formats as they are binary.)

`-h, --header`
: add header line with field names. Applicable only for text files (e.g. CSV) and XLSX file. When used with file formats Avro, JSON, Parquet, QVD, QVX or XML, warning is displayed and no header is written. It doesn't make sense for these formats.

`--header-file=<file>`
: add `<file>` before the first record When used with file formats Parquet, QVD, QVX or XLSX, warning is displayed and no `<file>` is prepended. (It doesn't make sense for these formats as they are binary.)

`--validate`
: without this option, no fields are checked against data types. With this option, all output fields are checked

`-x, --text-input`
: suppose the input as text, not binary

`--dos-eol`
: suppose the output is text with CRLF as end of line

`--mac-eol`
: suppose the output is text with CR as end of line

`-y, --text-output`
: write the output as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Options changing file suffix behaviour:

`--avro`
: whatever file's suffix, write the file in Avro file format

`--cmd=<cmd>`
: bash command `<cmd>` is used to write into `<file>`. In such case recognizing file's suffix is switched off. See examples below for inspiration.

`--csv`
: whatever file's suffix, write the file in as CSV using delimiters based on EVD (same as –text-output option)

`--gz`
: whatever file's suffix, use '`gzip`' to compress the file

`--ignore-suffix`
: ignore file's suffix, act only based on options

`--json`
: whatever file's suffix, write the file as JSON

`--parquet`
: whatever file's suffix, write the file in Parquet columnar file format

`--qvd`
: whatever file's suffix, write the file as Qlik's QVD file

`--qvx`
: whatever file's suffix, write the file as Qlik's QVX file

`--xml`
: whatever file's suffix, write the file as XML

`--xlsx`
: whatever file's suffix, write the file as MS Excel sheet

XML specific options:

`--document-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

`--record-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

`--vector-element-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

JSON specific options:

`--array-output`
: using this flag the json output would be an array or records, i.e. '`[{...},{...},...,{...}]`'

`--omit-null-fields`
: for other than JSON file is this option ignored. Check '`man evl writejson`' for details.

Examples

When password is needed in following examples, they are taken from $HOME/.evlpass file.

1. Write local CSV file in EVL graph (an EVS file):
   ```
   TARGET_FILE="/home/myself/file.csv"
   ...
   Map   FLOW1 FLOW2  evd/f1.evd evd/f2.evd evm/f.evm
   Write FLOW2 $TARGET_FILE evd/f2.evd
   ```
2. Write JSON file to AWS S3 bucket:
   ```
   TARGET_FILE="s3://mybucket/file.json"
   ...
   Map   FLOW1 FLOW2  evd/f1.evd evd/f2.evd evm/f.evm
   Write FLOW2 $TARGET_FILE evd/f2.evd
   ```
3. Write Parquet file to Hadoop file system:
   ```
   TARGET_FILE="hdfs:///some/path/file.parquet"
   ...
   Map   FLOW1 FLOW2  evd/f1.evd evd/f2.evd evm/f.evm
   Write FLOW2 $TARGET_FILE evd/f2.evd
   ```
4. Load gzipped CSV file to Google Storage:
   ```
   TARGET_FILE="gs://some_bucket/some/path/file.csv.gz"
   ...
   Map   FLOW1 FLOW2  evd/f1.evd evd/f2.evd evm/f.evm
   Write FLOW2 $TARGET_FILE evd/f2.evd
   ```
5. Load data to Postgres table:
   ```
   TARGET_FILE="postgres://tech_user@10.11.12.13:5432/my_database/my_table"
   ...
   Map   FLOW1 FLOW2  evd/f1.evd evd/f2.evd evm/f.evm
   Write FLOW2 $TARGET_FILE evd/f2.evd
   ```
6. Example of standalone usage: Write gzipped CSV file with header and validated data types over SFTP to some server:
   ```
   evl write -d 'id int sep=";", value string sep="\n"' --header -xy --validate \
         sftp://my_user@10.11.12.13:22/some/path/example.csv.gz < example.csv
   ```

12.2 Writeevd *(since EVL 2.5)*

Read EVD from stdin using this evd structure:

```
parents vector null=""
  string
name string
data_type string
format string null=""
comment string null=""
null vector null=""
  string
separator string null=""
quote struct null=""
  char string(1)
  optional uchar
options vector
```

```
        struct
          tag string
          value string null=""
      decimal struct null=""
        precision uchar
        scale uchar
        decimal_separator string(1) null=""
        thousands_separator string null=""
      string struct null=""
        length ulong null=""
        locale string null=""
        encoding string null=""
        max_bytes ulong null=""
        max_chars ulong null=""
      ustring struct null=""
        length ulong null=""
        locale string null=""
        encoding string null=""
        max_bytes ulong null=""
        max_chars ulong null=
```

`Writeevd`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl writeevd`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writeevd
  <f_in> <f_out> [-x|--text-input]

evl writeevd
  [-x|--text-input]
  [-v|--verbose]

evl writeevd
  ( --help | --usage | --version )
```

Options

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.3 Writejson *(since EVL 1.2)*

Write to stdout or `<f_out>` JSON formatted text where all fields exist (unless '`-n`' option) and are in order as defined in `<evd>`.

`Writejson`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl writejson`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writejson
  <f_in> <f_out> (<evd>|-d <inline_evd>)
  [-a|--array-output]
  [-n|--omit-null-fields] [-x|--text-input]

evl writejson
  (<evd>|-d <inline_evd>)
  [-a|--array-output]
  [-n|--omit-null-fields] [-x|--text-input]
  [-v|--verbose]

evl writejson
  ( --help | --usage | --version )
```

Options

`-a, --array-output`
: using this flag the json output would be an array or records, i.e. '`[{...},{...},...,{...}]`'

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'user_sum long'

`-n, --omit-null-fields`
: by this option, null fields are not presented in the output

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.4 Writekafka *(since EVL 1.1)*

Component calls kafka producer command, specified by ‘`EVL_KAFKA_PRODUCER_COMMAND`’, which is by default ‘`kafka-console-producer.sh`’. and run it with options:

```
--bootstrap-server "<server:port>,<server2:port2>,..." \
--topic "<topic>" ``<kafka_producer_opts>``
```

and send there `<f_in>`.

`Writekafka`
: is to be used in EVS job structure definition file. `<f_in>` is either output file or flow name.

`evl writekafka`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVS is EVL job structure definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writekafka
  <f_in> <topic>
  -s|--bootstrap-server <server:port>[,<server2:port2>...]
  [<kafka_producer_opts>]

evl writekafka
  <topic>
  -s|--bootstrap-server <server:port>[,<server2:port2>...]
  [<kafka_producer_opts>]
  [-v|--verbose]

evl writekafka
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.5 Writemysql *(since EVL 2.4)*

Write stdin or `<f_in>` into `<table>` of MariaDB/MySQL. If the table is not empty, it is truncated unless "–append" option is used.

Password is taken from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`'. When such file has not permissions 600 (or 400), it is ignored! For details see '`evl-password`'.

`Writemysql`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writemysql`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writemysql
  <f_in> [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-a|--append]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<mysqluser>] [--mysql=<mysql-options>] [-x|--text-input]

evl writemysql
  [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-a|--append]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<mysqluser>] [--mysql=<mysql-options>] [-x|--text-input]
  [-v|--verbose]

evl writemysql
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: -d 'id int, name string, started timestamp'

`-a, --append`
: target table is appended, not truncated

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

'mysql' options:

`-b, --dbname=<database>`
: this option is provided to '`mysql`' command as '`--database=<database>`'

`-h, --host=<hostname>`
: this option is provided to '`mysql`' command

`-p, --port=<port>`
: using other than standard port 3306. This option is provided to '`mysql`' command.

`-u, --username=<mysqluser>`
: if not mentioned, then current system username is used as mysql user. This option is provided to '`mysql`' command as '`--user=<mysqluser>`'.

`--mysql=<mysql-options>`
: other mysql options can be specified here

12.6 Writeora *(since EVL 2.2)*

Write stdin or `<f_in>` into `<table>` in Oracle database. If the table is not empty, it is truncated unless '`--append`' option is used. When delete statement need to be used instead of truncate, use option '`--delete`'.

When `<schema>` is not present, environment variable '`ORADATABASE`' is used.

Password is taken from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`'. When such file has not permissions 600 (or 400), it is ignored! For details see '`evl-password`'.

`Writeora`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writeora`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

`SQL*Loader Field Separator:`
: Writing the table by SQL*Loader uses as field seprator the value of '`$EVL_ORACLE_FIELD_SEPARATOR`', which is by default set to '`\x1f`' (i.e. an Unit Separator), and last field in each record is separated by '`\n`'.

`SQL*Loader script options`
: Custom options might be added to SQL*Loader script by environment variable '`$EVL_ORACLE_SQLLDR_OPTIONS`', which can contain comma or newline separated list of options like:

```
BINDSIZE = n
COLUMNARRAYROWS = n
DATE_CACHE = n
DEGREE_OF_PARALLELISM = {degree-num|DEFAULT|AUTO|NONE}
DIRECT = {TRUE | FALSE}
EMPTY_LOBS_ARE_NULL = {TRUE | FALSE}
ERRORS = n
```

```
EXTERNAL_TABLE = {NOT_USED | GENERATE_ONLY | EXECUTE}
FILE = tablespace file
LOAD = n
MULTITHREADING = {TRUE | FALSE}
PARALLEL = {TRUE | FALSE}
READSIZE = n
ROWS = n
SDF_PREFIX = string
SILENT = {HEADER | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL}
SKIP = n
SKIP_INDEX_MAINTENANCE = {TRUE | FALSE}
SKIP_UNUSABLE_INDEXES = {TRUE | FALSE}
STREAMSIZE = n
TRIM = {LRTRIM|NOTRIM|LTRIM|RTRIM|LDRTRIM}
```

and by default the variable is defined as:

```
export EVL_ORACLE_SQLLDR_OPTIONS="DIRECT = TRUE"
```

Synopsis

```
Writeora
  <f_in> [<schema>.]<table> <evd> [-x|--text-input]
  [-a|--append | --delete] [-u|--username=<oracle_user>]
  [ --connect=<connect_identifier> | -b|--dbname=<database> -h|--host=<hostname> [-p|-
  [--reject=<f_out>] [--control=<ctl_file>]

evl writeora
  [<schema>.]<table> <evd> [-x|--text-input]
  [-a|--append | --delete] [-u|--username=<oracle_user>]
  [ --connect=<connect_identifier> | -b|--dbname=<database> -h|--host=<hostname> [-p|-
  [--reject=<f_out>] [--control=<ctl_file>]
  [-v|--verbose]

evl writeora
  ( --help | --usage | --version )
```

Options

`-a, --append`
: with this option data will be added to the table, otherwise overwrite it

`--control=<ctl_file>`
: to use other than generated control file for SQL*Loader

`--delete`
: with this option data will be deleted, not truncated

`--reject=<f_out>`
: to catch the '`BADFILE`' file from SQL*Loader

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

’sqlldr’ options:

`--connect=<connect_identifier>`
: sqlldr will be called in the form:

```
<username>/<password>@<connect_identifier>
```

where <connect_identifier> can be in the form:

```
[<net_service_name> | [//]Host[:Port]/<service_name>]
```

without this option environment variable ‘`ORACONN`’ (if defined) is used as connection identifier for sqlldr

`-b, --dbname=<database>`
: either this or environment variable ‘`ORADATABASE`’ should be provided, If also ‘`ORADATABASE`’ environment variable is set, this option has preference.

`-h, --host=<hostname>`
: either this or environment variable ‘`ORAHOST`’ should be provided when connecting to other host than localhost. If also ‘`ORAHOST`’ variable is set, this option has preference.

`-p, --port=<port>`
: either this or environment variable ‘`ORAPORT`’ should be provided when using other than standard port ‘`1521`’.

`-u, --username=<oracle_user>`
: without this option environment variable ‘`ORAUSER`’ is used as user for sqlldr

12.7 Writeparquet *(since EVL 2.0)*

Write stdin or `<f_in>` into `<parquet>` directory as files of the size approximately of the size `<file_size>` MB. Compression can be turned on by –compression option.

`Writeparquet`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writeparquet`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writeparquet
  <f_in> <parquet> (<evd>|-d <inline_evd>) [-x|--text-input]
  [--compression=(gzip|snappy|lz4|brotli|zstd)]
  [--size=<file_size>] [--impala]

evl writeparquet
```

```
    <parquet> (<evd>|-d <inline_evd>) [-x|--text-input]
    [--compression=(gzip|snappy|lz4|brotli|zstd)]
    [--size=<file_size>] [--impala]
    [-v|--verbose]

  evl writeparquet
    ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: -d 'id int, name string, started timestamp'

`--compression=<compression>`
: compression to be used, possible values are gzip, snappy, lz4, brotli, zstd, none. By default 'none' is used, so no compression is applied

`--size=<file_size>`
: specify the number of MB, this size will be used for resulting files, default is 256 MB

`--impala`
: produce a parquet file(s) to be used then by Apache Impala, i.e. store TIMESTAMP as INT96

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.8 Writepg *(since EVL 1.3)*

Write stdin or `<f_in>` into `<table>` of PostgreSQL. If the table is not empty, it is truncated unless "–append" option is used.

Password is taken:

1. from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`',
2. from file '`$PGPASSFILE`', which is by default '`$HOME/.pgpass`'.

When such file has not permissions 600, it is ignored! For details see '`evl-password`'.

`Writepg`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writepg`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writepg
  <f_in> [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-a|--append]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>] [-x|--text-input]

evl writepg
  [<schema>.]<table> (<evd>|-d <inline_evd>)
  [-a|--append]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>] [-x|--text-input]
  [-v|--verbose]

evl writepg
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: ‘`-d 'id int, name string, started timestamp'`’

`-a, --append`
: target table is appended, not truncated

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

’psql’ options:

`-b, --dbname=<database>`
: either this or environment variable ‘`PGDATABASE`’ should be provided, if not, then current system username is used as psql database. If also ‘`PGDATABASE`’ environment variable is set, this option has preference. (This option is provided to ‘`psql`’ command.)

`-h, --host=<hostname>`
: either this or environment variable ‘`PGHOST`’ should be provided when connecting to other host than localhost. If also ‘`PGHOST`’ variable is set, this option has preference. (This option is provided to ‘`psql`’ command.)

`-p, --port=<port>`
: either this or environment variable ‘`PGPORT`’ should be provided when using other than standard port ‘`5432`’. (This option is provided to ‘`psql`’ command.)

`--psql=<psql_options>`
: all other options to be provides to psql command. See ‘`man psql`’ for details.

`-u, --username=<user>`
: either this or environment variable ‘`PGUSER`’ should be provided, if not, then current system username is used as psql user. If variable ‘`PGUSER`’ is set, this option has preference. (This option is provided to ‘`psql`’ command.)

12.9 Writeqvd *(since EVL 2.2)*

Write standard input or `<f_in>` into `<file.qvd>`.

`Writeqvd`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writeqvd`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writeqvd
  <f_in> <file.qvd> (<evd>|-d <inline_evd>)
  [-x|--text-input | -a|--text-input-dos-eol | -b|--text-input-mac-eol]
  [-s|--skip-bom] [-n|--null-as-string[=<string>]]

evl writeqvd
  <file.qvd> (<evd>|-d <inline_evd>)
  [-x|--text-input | -a|--text-input-dos-eol | -b|--text-input-mac-eol]
  [-s|--skip-bom] [-n|--null-as-string[=<string>]]
  [-v|--verbose]

evl writeqvd
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: ‘`-d 'id int, name string, started timestamp'`’

`-n, --null-as-string[=<string>]`
: write `<string>` instead of NULL value, without `<string>` specified it writes an empty string instead of NULL

`--skip-bom`
: skip utf-8 BOM (Byte order mark) from the beginning of input, i.e. EF BB BF. Windows usually add it to files in UTF8 encoding

`-x, --text-input`
: suppose the input as text, not binary

`--text-input-dos-eol`
: suppose the input as text with CRLF as end of line

`--text-input-mac-eol`
: suppose the input as text with CR as end of line

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.10 Writeqvx *(since EVL 2.2)*

Write standard input or `<f_in>` into `<file>`.

`Writeqvx`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writeqvx`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writeqvx
  <f_in> <file> (<evd>|-d <inline_evd>) [-x|--text-input]

evl writeqvx
  <file> (<evd>|-d <inline_evd>) [-x|--text-input]
  [-v|--verbose]

evl writeqvx
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: '`-d 'id int, name string, started timestamp'`'

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.11 Writesqlite *(since EVL 2.7)*

Write stdin or `<f_in>` into SQLite `<table>`. If the table is not empty, it is truncated unless '`--append`' option is used.

Path to the database file is taken from environment variable '`$EVL_SQLITE_DATABASE`', unless `<db_file>` is specified.

`Writesqlite`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writesqlite`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
     Writesqlite
       <f_in> [<database>.]<table> (<evd>|-d <inline_evd>)
       [-x|--text-input] [-a|--append]
       [--dbname=<db_file>]

     evl writesqlite
       [<database>.]<table> (<evd>|-d <inline_evd>)
       [-x|--text-input] [-a|--append]
       [--dbname=<db_file>]
       [-v|--verbose]

     evl writesqlite
       ( --help | --usage | --version )
```

Options

`-a, --append`
: target table is appended, not truncated

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: -d 'id int, name string, started timestamp'

`--dbname=<db_file>`
: path to the SQLite database file; if this option is not used, database file is taken from environment variable '`$EVL_SQLITE_DATABASE`'.

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.12 Writetd *(since EVL 1.1)*

Write stdin or `<f_in>` into `<table>` of Teradata.

`WriteTD`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writetd`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
WriteTD
  <f_in> <database>.<table> (<evd>|-d <inline_evd>)
  [-a|--append] [-x|--text-input]

evl writetd
  <database>.<table> (<evd>|-d <inline_evd>)
  [-a|--append] [-x|--text-input]
  [-v|--verbose]

evl writetd
  ( --help | --usage | --version )
```

Options

`-a, --append`
: target table is appended, not truncated

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'id int, name string, started timestamp'

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.13 Writexlsx *(since EVL 2.1)*

Write stdin or `<f_in>` into `<file>`.

`Writexlsx`
: is to be used in EVS job structure definition file. `<f_in>` is either input file or flow name.

`evl writexlsx`
: is intended for standalone usage, i.e. to be invoked from command line and reading records from standard input.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Header and footer format:

All possible (self-explained) formats:

```
font:bold
font:italic
underline:single
underline:double
align:left
align:center
align:right
color:black
color:blue
color:brown
color:cyan
color:gray
color:green
color:lime
color:magenta
color:navy
color:orange
color:pink
color:purple
color:red
color:silver
color:white
```

```
color:yellow
bg-color:black
bg-color:blue
bg-color:brown
bg-color:cyan
bg-color:gray
bg-color:green
bg-color:lime
bg-color:magenta
bg-color:navy
bg-color:orange
bg-color:pink
bg-color:purple
bg-color:red
bg-color:silver
bg-color:white
bg-color:yellow
```

Synopsis

```
Writexlsx
  <f_in> <file> (<evd>|-d <inline_evd>)
  [-h|--header [<header_fields>] [--header-format]]
  [-f|--footer  <footer_fields>  [--footer-format]]
  [-x|--text-input]

evl writexlsx
  <file> (<evd>|-d <inline_evd>)
  [-h|--header [<header_fields>] [--header-format]]
  [-f|--footer  <footer_fields>  [--footer-format]]
  [-x|--text-input]
  [-v|--verbose]

evl writexlsx
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented. Example: -d 'id int, name string, started timestamp'

`-f, --footer=<footer_fields>`
: semicolon separated list of footer cells, it can be a string or if it begins with '=' sign it is a formula. Variable '`$COLUMN_RANGE`' can be used in such formulas which will be replaced by range of given column. Example: –footer 'Results:;=SUM($COLUMN_RANGE);AVERAGE($COLUMN_RANGE)'

`--footer-format=<footer_format>`
: semicolon separated format of the footer. Example: –footer-format "font:bold;bg-color:green;color:white"

`-h, --header=<header_fields>`
: semicolon separated list of header captions, empty `<header_fields>` means to use field names from EVD. Example: –header 'ID;timestamp;price'

`--header-format=<header_format>`
: semicolon separated format of the header. Example: –header-format "font:bold;color:red"

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

12.14 Writexml *(since EVL 1.3)*

Write to stdout or `<f_out>` XML formatted text where all fields exist and are in order as defined in `<evd>`.

`Writexml`
: is to be used in EVS job structure definition file. `<f_out>` is either output file or flow name.

`evl writexml`
: is intended for standalone usage, i.e. to be invoked from command line and and write to standard output.

EVD and EVS are EVL definition files, for details see evl-evd(5) and evl-evs(5).

Synopsis

```
Writexml
  <f_in> <f_out> (<evd>|-d <inline_evd>) [-x|--text-input]
  [--document-tag=<tag>] [--record-tag=<tag>] [--vector-element-tag=<tag>]

evl writexml
  (<evd>|-d <inline_evd>) [-x|--text-input]
  [--document-tag=<tag>] [--record-tag=<tag>] [--vector-element-tag=<tag>]
  [-v|--verbose]

evl writexml
  ( --help | --usage | --version )
```

Options

`-d, --data-definition=<inline_evd>`
: either this option or the file <evd> must be presented. Example: -d 'user_sum long'

`--document-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

`--record-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

`--vector-element-tag=<tag>`
: for other than XML file is this option ignored. Check '`man evl writexml`' for details.

`-x, --text-input`
: suppose the input as text, not binary

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

13 Commands

EVL jobs are defined in `evs` files and consist of components connected by flows (pipes). But there are also several commands which (mostly) follow standard Unix commands or bash build-in functions, but operates according to URI, like '`hdfs://`' for Hadoop, '`gs://`' for Google Storage, '`s3://`' for Amazon S3 storage or '`sftp://`' for files to be accessed over SSH.

These commands can be used either in EVL jobs or in EVL workflows and also as a command line utilities.

File and directory manipulation

(Behaves by URI, i.e. '`hdfs://`', '`gs://`', '`s3://`', '`sftp://`')

- Section 13.3 [Cp], page 141,
- Section 13.4 [Chmod], page 143,
- Section 13.8 [Ls], page 147,
- Section 13.11 [Mkdir], page 151,
- Section 13.12 [Mv], page 152,
- Section 13.14 [Rm], page 155,
- Section 13.15 [Rmdir], page 156,
- Section 13.22 [Test], page 170,
- Section 13.23 [Touch], page 170,

EVL job and workflow specific commands

- Section 13.1 [Calendar], page 139,
- Section 13.6 [End], page 145,
- Section 13.9 [Mail], page 148,
- Section 13.13 [Project], page 153,
- Section 13.16 [Run], page 157,
- Section 13.19 [Sleep], page 167,
- Section 13.20 [Spark], page 168,
- Section 13.24 [Wait], page 171,

EVL task manipulation commands

- Section 13.2 [Cancel], page 140,
- Section 13.7 [Log], page 146,
- Section 13.17 [Set], page 165,
- Section 13.18 [Skip], page 166,
- Section 13.21 [Status], page 169,

Other commands

- Section 13.5 [Crontab], page 144,
- Section 13.10 [Manager], page 150,

13.1 Calendar *(since EVL 2.8)*

Based on specified calendar file(s) EVL Calendar command either:

- continue processing of EVL job or workflow, or
- successfully end the task.

It compares value of '`EVL_ODATE`' (Order Date) with dates in calendar file(s).

Calendar file is simply a text file with a list of dates in format '`YYYY-MM-DD`', one date per line.

When more than one calendar is specified, they are concatenated, so behaves as logical OR.

To achive logical AND simply user several calendar commands.

Unless '`--blacklist`' option is used, whitelist of dates is assumed.

There are these predefined calendar files:

- '`first_day_of_month.cal`'
- '`last_day_of_month.cal`'
- '`workday.cal`' - i.e. Monday to Friday
- '`weekend.cal`'
- '`monday.cal`'
- '`tuesday.cal`'
- '`wednesday.cal`'
- '`thursday.cal`'
- '`friday.cal`'
- '`saturday.cal`'
- '`sunday.cal`'

These calendar files can be specified with no path, just a name. You can find them usually in '`/opt/evl/share/templates/calendar/`'

Unless '`--project`' option is used, it search in current project directory in '`calendar`' subdirectory. Then it search for common calendars (usually) in '`/opt/evl/share/templates/calendar/`'.

`Calendar`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl calendar`
: is intended for standalone usage, i.e. to be invoked from command line. In this case when the '`EVL_ODATE`' do not match, the command exit with error code 1.

Synopsis

```
Calendar
  <calendar>.cal [<calendar2.cal>...]
  [--blacklist] [-p|--project=<project_dir>]
  [--no-end]

evl calendar
  <calendar>.cal [<calendar2.cal>]
  [--blacklist] [-p|--project=<project_dir>]
  [-v|--verbose]
```

```
evl calendar
  ( --help | --usage | --version )
```

Options

`--blacklist`
: to blacklist dates from calendar file(s)

`--no-end`
: do not end a workflow in case of no calendar match. In such case can be used in condition. See examples.

`-p, --project=<project_dir>`
: specify project folder to search for calendar file(s) in '`calendar`' subfolder

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To run a workflow every working day in Czechia, put this at the beginning of EWS file:

   ```
   Calendar workday.cal
   Calendar --blacklist czech_holidays.cal
   ```

   where '`calendar/czech_holidays.cal`' supposed to be in the same project as the EWS file.
2. To run an EVL job only on Mondays, Wednesdays, and Fridays, add this line at the beginning of EVS job structure file:

   ```
   Calendar monday.cal wednesday.cal friday.cal
   ```
3. To run an EVL job only on last day of a month, but then continue processing a workflow:

   ```
   if Calendar --no-end last_day_of_month.cal
   then
     Run job/last_day_of_month.evl
   fi
   Run job/as_usual.evl
   ```

13.2 Cancel *(since EVL 2.4)*

To cancel running EVL task (i.e. job, workflow or script, or waiting for a file). Either by Run ID or by a task name and Order Date.

When there are several task(s) with given name, it tries to cancel the latest one.

It recognize type of task based on the file suffix.

`*.evl`
: EVL job

`*.ewf`
: EVL workflow

`*.sh`
: bash script

`Any other or no suffix`
: file mask of file(s) to waiting for

Synopsis

```
evl cancel
  ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project_dir>] [-v|--verbose]

evl cancel
  ( --help | --usage | --version )
```

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To cancel a job with Run ID 145:
   ```
   evl cancel 145
   ```
2. To cancel jobs of Run IDs between 145 and 150:
   ```
   evl cancel {145..150}
   ```
3. To cancel a workflow '`billing.ewf`' of project '`/data/project/billing`' with yesterday Order Date:
   ```
   evl cancel --odate=yesterday billing.ewf --project=/data/project/billing/
   ```

13.3 Cp *(since EVL 1.3)*

Copy `<source>` to `<dest>`, or multiple `<source>`(s) to `<dest>` directory. `<source>` and `<dest>` might be one of:

```
<local_path>
```

```
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

On local files it calls standard 'cp' command to copy files, but when `<source>` and/or `<dest>` contain URI '`hdfs://`' (i.e. Hadoop FS), then it calls appropriate commands to copy from/to such source/destination.

For example when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_cp`', which is by default '`aws s3 cp`'.

So far it can copy to/from local path from/to some specific one or within one URI type. So not yet possible to copy dirrectly for example from S3 to G-Drive.

When more than one `<source>` is specified, then URI prefix must be the same for all of them.

`Cp`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl cp`
: is intended for standalone usage, i.e. to be invoked from command line.

Using this command might keep your code clean, but for more complex copying better use appropriate commands directly, as it gives you all the available options for particular storage or protocol type.

Synopsis

```
Cp
  [-f|--force] [-p] <source>... <dest>

evl cp
  [-f|--force] [-p] <source>... <dest>
  [--verbose]

evl cp
  ( --help | --usage | --version )
```

Options

`-f, --force`
: destination will be overwritten if exists

`-p`
: preserve mode (i.e. permission), timestamps and ownership

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
print version and exit

Examples

These lines in EVL job (an 'evs' file):

```
Cp hdfs:///some/path/to/file /some/local/path/
Cp /some/local/path/file hdfs:///some/path/
Cp hdfs:///some/path/file hdfs:///other/path/
Cp /some/local/path/file /other/local/path/
```

will call (with handling fails in proper EVL way):

```
evl_hdfs_get hdfs:///some/path/file /some/local/path/
evl_hdfs_put /some/local/path/file hdfs:///some/path/
evl_hdfs_cp hdfs:///some/path/file hdfs:///other/path/
cp /some/local/path/file /other/local/path/
```

where defaults for '`evl_hdfs_*`' variables are:

```
function evl_hdfs_get { hdfs dfs -get ; }
function evl_hdfs_cp  { hdfs dfs -cp  ; }
function evl_hdfs_put { hdfs dfs -put ; }
```

13.4 Chmod *(since EVL 2.3)*

Change file mode bits.

Each `<file>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_chmod`', which is by default '`hdfs dfs -chmod`'.

For scheme '`sftp://`' it calls function '`evl_sftp_chmod`'.

Synopsis

```
Chmod
  ( <mode>[,<mode>]... | <octal-mode> ) <file>...
  [-R|--recursive]

evl chmod
  ( <mode>[,<mode>]... | <octal-mode> ) <file>...
  [-R|--recursive] [--verbose]

evl chmod
  ( --help | --usage | --version )
```

Options

`-R, --recursive`
change files and directories recursively

Standard options:

`--help`
print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Simple usage examples:

   ```
   Chmod hdfs:///some/path/
   Chmod /some/local/machine/path/
   ```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

   ```
   # on DEV:
   OUTPUT_DIR=/data/output
   # on PROD:
   OUTPUT_DIR=hdfs:///data/output
   ```

   and then in '`evs`' file:

   ```
   Chmod -p "$OUTPUT_DIR"
   ```

13.5 Crontab *(since EVL 2.4)*

Create/update/remove EVL section of given EVL project of current user's crontab based on '`crontab.sh`' file from current directory or from `<project_dir>`.

Synopsis

```
evl crontab
  ( set | get | remove )
  [-p|--project=<project_dir>] [-v|--verbose]

evl crontab
  ( --help | --usage | --version )
```

Options

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. There is a ‘`crontab.sh`’ file in the EVL project created by ‘`evl project new`’ or ‘`evl project sample`’. It is usually a good start.
2. Example of ‘`crontab.sh`’ file:

```
# crontab.sh -- schedule EVL project workflows
#
# This file is processed by 'evl crontab set' command to create/update crontab entr
#

# Schedule workflows per environment
case "$EVL_ENV" in
  DEV)
    # Run workflow/sample.ewf Mo-Fr at 8:10, 10:10, 12:10, ..., 16:10
    Schedule 10 8-16/2 * * 1-5 sample.ewf --odate yesterday
  ;;
  TEST)
    # Run workflow/sample.ewf Mo-Fr at 5:10am
    Schedule 10 5 * * 1-5 sample.ewf
    # Restart (with the same Order Date) Mo-Fr at 6:10am
    Schedule --restart 10 6 * * 1-5 sample.ewf --odate yesterday
  ;;
  PROD)
    # Run workflow/sample.ewf daily at 5:10am
    Schedule 10 5 * * * sample.ewf --odate yesterday
    # Restart (with the same Order Date) at 6:10am
    Schedule --restart 10 6 * * * sample.ewf --odate yesterday
  ;;
esac
```

13.6 End *(since EVL 2.0)*

Finish processing the EVL job or workflow, ignoring the rest of the EVS/EWS file.

EVS is EVL job definition file, for details see evl-evs(5). EWS is EVL workflow definition file, for details see evl-ews(5).

Synopsis

```
End

evl end
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`

print this help and exit

`--usage`

print short usage information and exit

`--version`
: print version and exit

Examples

1. EVL job (an EVS file) might end like this:

```
...
Write  FLOW_99 /path/to/file evd/sample.evd --text-output
End
Here can be any comment, notes or pieces of code...
```

2. EVL workflow (an EWS file) might end like this:

```
...
Run some_job.evl
End
Here can be any comment, notes or pieces of code...
```

13.7 Log *(since EVL 2.4)*

Get status log entries of EVL task(s) (i.e. job, workflow or script, or waiting for a file). When `<regex>` ends with '`.evl`', '`.ewf`' or '`.sh`', it looks for appropriate task type.

Synopsis

```
evl log
  <regex>... [--state=<state>] [-o|--odate=<odate_regex>]
  [-p|--project=<project_dir>] [-v|--verbose]

evl log
  <run_id>... [-p|--project=<project_dir>] [-v|--verbose]

evl log
  ( --help | --usage | --version )
```

Options

`-o, --odate=<odate>`
: to specify particular Order Date

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--state=<state>`
: to specify particular state, possible are '`reru`', '`wait`', '`runn`', '`fail`', '`canc`', '`skip`', '`succ`', '`arch`', '`dele`'

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
  print version and exit

Examples

1. To get information about tasks with Run ID between 145 and 150:
   ```
   evl log {145..150}
   ```
2. To get information about a workflow '`billing.ewf`' of project '`/data/project/billing`':
   ```
   evl log billing --project=/data/project/billing/ | grep "|ewf|"
   ```
3. To get all successfully finished jobs with OrderDate in 01/2026:
   ```
   evl log ".*" --odate "202601.." --state succ
   ```

13.8 Ls *(since EVL 2.0)*

List `<dest>`, which might be one of:

```
<local_path>
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

So for example when argument starts with '`hdfs://`', then it is supposed to be on HDFS file system and calls the function '`evl_hdfs_ls`', which is by default '`hadoop fs -ls`'.

Or when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_ls`', which is by default '`aws s3 ls`'.

Otherwise act as usual '`ls`' command.

`Ls`
  is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl ls`
  is intended for standalone usage, i.e. to be invoked from command line.

Synopsis

```
Ls
  <dest>...
  [--force]
  [-Q|--quote-name]
  [-r|-R|--recursive]
  [-z|--zero]

evl ls
  <dest>...
  [--force]
  [-Q|--quote-name]
  [-r|-R|--recursive]
  [-z|--zero]
  [--verbose]
```

```
evl ls
  ( --help | --usage | --version )
```

Options

`--force`
: do not fail if <dest> doesn't exist

#-m, –comma-separated # produce a comma separated list of entries

`-Q, --quote-name`
: enclose entry names in double quotes

`-r, -R, --recursive`
: list subdirectories recursively

`-z, --zero`
: line delimiter is NUL, not newline

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. These simple examples write result on stdout:

   ```
   Ls hdfs:///some/path/????-??-??.csv
   Ls s3:///somebucketname/path/
   Ls /some/local/machine/path/*
   ```

2. To be used to initiate a flow in EVL job:

   ```
   INPUT_FILES=/data/input
   Run   ""    INPUT  "Ls $INPUT_FILES"
   Map   INPUT ...
   ...
   ```

And then, for PROD environment, input files would be defined for example:

```
INPUT_FILES=hdfs:///data/input
```

13.9 Mail *(since EVL 2.1)*

When no <email> is specified, the default (comma separated list of) recipients are taken from environment variable '`EVL_MAIL_TO`'.

When environment variable '`EVL_MAIL_SEND`' is set to 0, then no e-mails are sent by this command. Useful to be set for non-production environments.

'`Mail`' command will call command from environment variable '`EVL_MAIL`' which suppose to be standard Unix command **mail** with possible other parameters.

```
Mail
```

is to be used either in EVL Job or in EVL Workflow structure definition file, i.e. in ‘evs’ or ‘ews’ file.

`evl mail`
: is intended for standalone usage, i.e. to be invoked from command line.

EVS is EVL job definition file and EWS is EVL Workflow definition file, for details see man evl-evs(5) or ewf-ews(5).

Synopsis

```
Mail
  <subject> <message> [ <email>[,...] ]
  [-a <attachment>]... [-c <cc_email>[,...]] [-b <bcc_email>[,...]]

evl mail
  <subject> <message> [ <email>[,...] ]
  [-a <attachment>]... [-c <cc_email>[,...]] [-b <bcc_email>[,...]]

evl mail
  ( --help | --usage | --version )
```

Options

`-a <attachment>`
: attach the given file to the message, can be used several times to add more files

`-b <bcc_email>[,...]`
: send blind carbon copies

`-c <cc_email>[,...]`
: send carbon copies

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Following setting will e-mail carbon copy of every e-mail to ‘admin@server’:

   ```
   export EVL_MAIL='mail -c "admin@server"'
   ```

   Following invocation of "Mail":

   ```
   Mail "Job Failed" "Job extract_file_billing failed."
   ```

   will actually run:

   ```
   echo "Job extract_file_billing failed." | \
       mail -c "admin@server" -s "Job Failed" "$EVL_MAIL_TO"
   ```

   and log appropriate information into EVL or EWF log.

2. For non-production environment it worth to set:
   ```
   export EVL_MAIL_SEND=0
   ```
   so then in example 1. you will obtain only such a warning in a log file:
   ```
   EVL_MAIL_SEND is set to 0, so no e-mail was sent to \
      "$EVL_MAIL_TO" with subject "Job Failed".
   ```

13.10 Manager *(since EVL 2.4)*

To manipulate monitor database of given `<project>` used by EVL Manager.

`update`
: to update EVL Manager database based on EVL status log. When no project is specified, suppose current directory as a project folder.

Synopsis

```
evl manager user
  add <username> [-p|--password=<password>] [-a|--admin] [-v|--verbose]

evl manager project
  add <project_name> [-p|--path=<project_path>] [-v|--verbose]

evl manager user-to-project
  <project_name> <username> [-v|--verbose]

evl manager update
  [-p|--project=<project_dir>] [-v|--verbose]

evl manager
  ( --help | --usage | --version )
```

Options

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To update monitor DB according to status log of project '`billing`':
   ```
   evl manager update -p billing
   ```

13.11 Mkdir *(since EVL 1.0)*

Create `<directory>`, which might be one of:

```
<local_path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
```

With option '`--parents`' no error if directory already exists and make parent directories as needed.

Each `<directory>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_mkdir`', which is by default '`hdfs dfs -mkdir`'.

For scheme '`s3://`' it calls function '`evl_s3_mkdir`'.

For scheme '`sftp://`' it calls function '`evl_sftp_mkdir`'.

Synopsis

```
Mkdir
  [-p|--parents] <directory>...

evl mkdir
  [-p|--parents] <directory>...

evl mkdir
  ( --help | --usage | --version )
```

Options

`-p, --parents`
no error if existing, make parent directories as needed

Standard options:

`--help`
print this help and exit

`--usage`
print short usage information and exit

`--version`
print version and exit

Examples

1. Simple usage examples:

```
Mkdir hdfs:///some/path/
Mkdir /some/local/machine/path/
```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

```
# on DEV:
OUTPUT_DIR=/data/output
# on PROD:
OUTPUT_DIR=hdfs:///data/output
```

and then in 'evs' file:

```
Mkdir -p "$OUTPUT_DIR"
```

13.12 Mv *(since EVL 1.0)*

Move <source> to <dest>, or multiple <source>(s) to <dest> directory. <source> and <dest> might be one of:

```
<local_path>
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

So for example when argument starts with 'hdfs://', then it is supposed to be on HDFS file system and calls the function 'evl_hdfs_mv', which is by default 'hadoop fs -mv'.

Or when argument starts with 's3://', then it is supposed to be on S3 file system and calls the function 'evl_s3_mv', which is by default 'aws s3 mv'.

Otherwise act as usual 'mv' command.

So far it can move to/from local path from/to some specific one or within one URI type. So not yet possible to move dirrectly for example from S3 to G-Drive.

#But when <source> and/or <dest> contain URI like 'hdfs://', 's3://', #'gs://' or 'sftp://' (i.e. Hadoop FS, Amazon S3, Google Storage and SFTP), #then it calls appropriate commands to move (or copy and delete) from/to such source/destination.

When more than one <source> is specified, then URI prefix must be the same for all of them.

`Mv`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl mv`
: is intended for standalone usage, i.e. to be invoked from command line.

Using this command might keep your code clean, but for more complex copying better use appropriate commands directly, as it gives you all the available options for particular storage or protocol type.

Synopsis

```
Mv
  [-f|--force] <source>... <dest>

evl mv
  [-f|--force] <source>... <dest>
  [-v|--verbose]

evl mv
  ( --help | --usage | --version )
```

Options

`-f, --force`
: destination will be overwritten if exists

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. This line in EVL job (an '`evs`' file):

   ```
   Mv hdfs:///some/path/to/file /some/local/path/
   ```

   will call:

   ```
   evl_hdfs_get hdfs:///some/path/file /some/local/path/
   evl_hdfs_rm hdfs:///some/path/file
   ```

2. This line in EVL job (an '`evs`' file):

   ```
   Mv /some/local/path/file hdfs:///some/path/
   ```

   will call:

   ```
   evl_hdfs_put /some/local/path/file hdfs:///some/path/
   rm /some/local/path/file
   ```

3. This line in EVL job (an '`evs`' file):

   ```
   Mv hdfs:///some/path/file hdfs:///other/path/
   ```

   will call:

   ```
   evl_hdfs_cp hdfs:///some/path/file hdfs:///other/path/
   evl_hdfs_rm hdfs:///some/path/file
   ```

4. This line in EVL job (an '`evs`' file):

   ```
   Mv /some/local/path/file /other/local/path/
   ```

   will call:

   ```
   mv /some/local/path/file /other/local/path/
   ```

Where defauls for '`evl_hdfs_*`' variables are:

```
function evl_hdfs_cp  { hdfs dfs -cp  ; }
function evl_hdfs_get { hdfs dfs -get ; }
function evl_hdfs_put { hdfs dfs -put ; }
function evl_hdfs_rm  { hdfs dfs -rm ; }
```

13.13 Project *(since EVL 1.0)*

Create new EVL project(s) or get project settings. Consider current directory as a project one, unless `<project_dir>` is specified with either full or relative path. Last folder in the `<project_dir>` path is considered as project name. Prefer to use small letters for project names, however numbers, capital letters, underscores and dashes are possible.

Projects can be included into another projects. But remember that parent's project.sh is not automatically included (i.e. sourced) by subproject's one.

`create <project_name> [<project_name_2>...]`

create <project_name> directory (directories) with standard subfolders structure and default '`project.sh`' configuration file.

`create --sample <project_name> [<project_name_2>...]`

create <project_name> directory (directories) with sample data and sample jobs and workflows.

`get <variable_name> [--path] [--omit-newline] [--project=<project_dir>]`

get the value of <variable_name>, based on the project.sh configuration file. Search '`project.sh`' in the current directory, unless <project_dir> if mentioned. With option '`--path`', it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning). With option '`--omit-newline`', return value without trailing newline.

To drop the whole project simply delete the folder recursively.

Synopsis

```
evl project create
  <project_name>... [--sample]
  [-v|--verbose]

evl project get
  <variable_name>
  [-p|--project=<project_dir>]
  [--path] [--omit-newline]
  [-v|--verbose]

evl project
  ( --help | --usage | --version )
```

Options

`--omit-newline`

return value without trailing newline, good for example for assigning returned value into a variable

`--path`

it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning)

`-p, --project=<project_dir>`

specify project folder if not the current working one

`--sample`

create project with sample configuration

Standard options:

`--help`

print this help and exit

`--usage`

print short usage information and exit

`-v, --verbose`

print to stderr info/debug messages of the component

`--version`
: print version and exit

Environment Variables

These variables can be set in the project's '`project.sh`' file. Here are mentioned with their default values.

`EVL_PROJECT_LOG_DIR="$EVL_LOG_PATH/<project_name>"`
: where all the project logs goes

`EVL_PROJECT_TMP_DIR="$EVL_TMP_PATH/<project_name>"`
: where all the temporary files should be placed for all tasks of the project

Examples

1. To create three main projects with couple of subprojects:

```
# shared to all projects
evl project create shared

evl project create stage       # shared stuff only for "stage" projects
evl project create stage/sap stage/tap stage/erp stage/signaling

evl project create dwh         # shared stuff only for "dwh" projects
evl project create dwh/usage dwh/billing dwh/party dwh/contract dwh/product

evl project create mart        # shared stuff only for "mart" projects
evl project create mart/marketing mart/sales
```

2. To create new project with sample data, jobs and workflows:

```
evl project create --sample my_sample
```

3. To get the project path to log directory (i.e. '`EVL_PROJECT_LOG_DIR`'):

```
evl project get --path EVL_PROJECT_LOG_DIR
```

13.14 Rm *(since EVL 2.0)*

Remove `<dest>`, which might be one of:

```
<local_path>
hdfs://<path>
gs://<bucket>/<path>
s3://<bucket>/<path>
sftp://<path>
```

So for example when argument starts with '`hdfs://`', then it is supposed to be on HDSF file system and calls the function '`evl_hdfs_rm`', which is by default '`hadoop fs -rm`'.

Or when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_rm`', which is by default '`aws s3 rm`'.

In all other cases standard '`rm`' command is used.

Synopsis

```
Rm
  [-f|--force] [-r|-R|--recursive] <dest>...

evl rm
```

```
    [-f|--force] [-r|-R|--recursive] <dest>...
    [--verbose]

  evl rm
    ( --help | --usage | --version )
```

Options

`-f, --force`
: ignore nonexistent files and arguments, never prompt

`-r, -R, --recursive`
: remove directories and their contents recursively

Examples

1. To remove file from HDFS:

```
Rm hdfs://some/path/to/file
```

13.15 Rmdir *(since EVL 2.6)*

Remove `<directory>` on local filesystem or on HDFS in case `<directory>` starts with '`hdfs://`' or on remote machine by '`ssh`' when starts with '`sftp://`'. It fails if the directories are empty.

Each `<directory>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_rmdir`', which is by default '`hdfs dfs -rmdir`'.

For scheme '`sftp://`' it calls function '`evl_sftp_rmdir`'.

Synopsis

```
  Rmdir
    [-p|--parents] <directory>...

  evl rmdir
    [-p|--parents] <directory>...

  evl rmdir
    ( --help | --usage | --version )
```

Options

`-p, --parents`
: remove `<directory>` and its ancestors, e.g. 'Rmdir -p a/b/c' is similar to 'Rmdir a/b/c a/b a'

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Simple usage examples:
   ```
   Rmdir hdfs:///some/path/
   Rmdir /some/local/machine/path/
   ```
2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:
   ```
   # on DEV:
   OUTPUT_DIR=/data/output
   # on PROD:
   OUTPUT_DIR=hdfs:///data/output
   ```
   and then use in '`evs`' file:
   ```
   Rmdir -p "$OUTPUT_DIR"
   ```
   and do not care if you operate locally or on HDFS.

13.16 Run *(since EVL 1.0)*

EVL Run command runs EVL task(s), i.e. `<job>`, `<workflow>`, or any shell `<script>`, or wait for a file to exist.

Tasks are provided as list of arguments or in a `<file.csv>` with '`--from-file`' option.

`Run`
: run EVL task(s) from EVL workflow (i.e. from an '`ews/*.ews`' file)

`evl run`
: is intended for standalone usage, i.e. to be invoked from command line

Type of the task is recognized by a file suffix:

`*.evl`
: suppose an EVL job, either full path or relative to project directory or relative to project's '`job/`' subfolder

`*.ewf`
: suppose an EVL workflow, either full path or relative to project directory or relative to project's '`workflow/`' subfolder

`*.sh`
: suppose a Bash script, either full path or relative to project directory or relative to project's '`job/`' subfolder

`w <file_mask>`
: in case of waiting for a file presence, wait flag '`w`' must be used followed by a file mask

If more than one task is provided, then run them in serial one after another, i.e. run the following one after previous successfully finished. In case of option '`--dependencies-file`' the order is taken from provided dependencies CSV file.

Failures and retries:

Once one EVL task fails, then whole '`Run`' or '`evl run`' command fails (unless '`$EVL_RUN_FAIL`' environment variable is set to 0, or option '`--ignore-fail`' is specified, but use with care).

In case of EVL task failure, '`Run`' or '`evl run`' command tries to restart it automatically '`$EVL_RUN_RETRY`' times. If `<retries>` is specified, then such value has precedence over '`$EVL_RUN_RETRY`'.

By option '`--run-on-fail`' a task to be run can be specified in case of a failure.

Maximal run time:

When the run time of given EVL task exceed '$EVL_RUN_TIME', then such task is killed and 'Run' or 'evl run' command fails. If <time> is specified, then such value has precedence over '$EVL_RUN_TIME'. Each retry measure run time from the beginning.

<time> can be specified in seconds, minutes, hours or days, so suffix 's', 'm', 'h' or 'd' need to be specified to the number. If no unit is specified, seconds are assumed.

Synopsis

```
Run
  ( [<time>[smhd]] [<retries>r] ( <job> | <workflow> | <script> | w <file_mask> ) )...
  [ --target k8s|local|ssh ]
  [ --ignore-fail ]
  [ --run-on-fail=<task> ]
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]

Run
  --from-file=<file.csv>
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]

evl run
  ( [<time>[smhd]] [<retries>r] ( <job> | <workflow> | <script> | w <file_mask>) )...
  [ --target k8s|local|ssh ]
  [ --ignore-fail ]
  [ --run-on-fail=<task> ]
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]
  [-s|--progress]
  [-v|--verbose]
  [ --monitor-db=<monitor_db_uri> ]
  [ --parent-run-id=<parent_run_id> ]

evl run
  ( --help | --usage | --version )
```

Options

`-c, --check-prev-run`
: check if given task(s) already finished for given Order Date and fail with exit code 2 if yes

`-D, --define=<definition>`
: the `<definition>` is evaluated right before running a task, but after evaluating settings from '`evl`' or '`ewf`' file, e.g. '`-DSOME_PATH=/some/path`' will do '`eval SOME_PATH=/some/path`', and overwrites then variable SOME_PATH possibly defined in '`evl`' or '`ewf`' file. Multiple '`--define`' options can be used.

`--dependencies-file=<file.csv>`
: read arguments from a CSV `<file.csv>`. CSV file (delimited by $EVL_CONFIG_FIELD_SEPARATOR, which is by default '`,`') must have a header and is of the form:

```
task_name;dependency_task_name;comment;rest_is_ignored
```

where:

`task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask.

`dependency_task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask.

`comment and the rest of the line`
: whatever for documentation purposes, is ignored. But no newlines!

`--from-file=<file.csv>`
: read arguments from a CSV `<file.csv>`. CSV file (delimited by $EVL_CONFIG_FIELD_SEPARATOR, which is by default '`,`') must have a header and is of the form:

```
task_name;max_time;retries;wait_for_file;target_type;nice;
    ignore_fail;run_task_on_fail;valid_from;valid_to;comment;rest_is_ignored
```

where:

`task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask. All (non wait-for-file) tasks must exist on filesystem, otherwise workflow fail.

`max_time`
: is maximal run time, see above

`retries`
: is the number of retries, see above

`wait_for_file`
: is '`Yes`'/'`No`' flag if the task_name is to be executed or is a file mask to wait for. If empty, '`No`' is assumed.

`target_type`
: possible values are '`local`' (default), '`ssh`' or '`k8s`'.

`nice`
: run task with this nice value, which is a number between 0 and 19. It is the standard Linux Nice value.

`ignore_fail`
: is '`Yes`'/'`No`' flag if failure of the task '`task_name`' can be ignored or not. Default is '`No`'.

`run_task_on_fail`
: run this task in case of failure of task '`task_name`'.

`valid_from`
: run task only if ODATE is greated than or equal to this value of the format YYYY-MM-DD. If empty, consider task valid.

`valid_to`
: run task only if ODATE is less than or equal to this value of the format YYYY-MM-DD. If empty, consider task valid.

`comment and the rest of the line`
: whatever for documentation purposes, is ignored. But no newlines!

`--ignore-fail`
: by default once some task to be run fails, the whole workflow fails (after other '`Run`' commands finish and reach first '`Wait`'). This option ignores this and allows other tasks specified by given '`Run`' command to finish. Also the whole workflow continues.

`--monitor-db=<monitor_db_uri>`
: specify Postgres DB to be used for monitoring

`-o, --odate=<yyyymmdd>`
: run evl job with specified Order Date, environment variable '`$EVL_ODATE`' is ignored

`-s, --progress`
: for an EVL job it shows the number of records passed each component, for an EVL workflow it shows the states of each component, and for running shell scripts it does nothin. The output is refreshed every '`$EVL_PROGRESS_REFRESH_SEC`' seconds. By default it is 2 seconds.

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--parent-run-id`
: for monitoring purpose in case of non-local targets, specifies under which workflow are EVL tasks invoked

`-r, --restart`
: do not continue given workflow(s), but restart them from the beginning

`--run-on-fail=<task>`
: when some EVL task fails, run this `<task>`. When used together with '`--ignore-fail`', then this '`run-on-fail <task>`' is fired immediately after the task fails and then continue with others. If also some other task fails, then this '`run-on-fail <task>`' is fired again.

`--target=( k8s | local | ssh )`
: run tasks on particular target, possible values are:

  `k8s`
  : run on Kubernetes cluster which is defined by '`$EVL_RUN_K8S_*`' variables

  `local`
  : run on local machine. This is the default value.

`ssh`
: run on a remote machine connected over ssh defined by '`$EVL_RUN_SSH_*`' variables

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Environment Variables

The list of variables which controls EVL Workflow '`Run`' or '`evl run`' command behaviour. With their default values. These variables can be set for example in user's '`~/.evlrc`' file or in the project's '`project.sh`'.

Control failures:

`EVL_RUN_FAIL=1`
: whether or not to fail given '`Run`' command once any EVL task fails, so when zero is set, the '`Run`' command continue regardless tasks failures

`EVL_RUN_FAIL_MAIL=1`
: whether or not to send an e-mail when the task fails

`EVL_RUN_FAIL_MAIL_SUBJECT='$EVL_PROJECT FAILED'`
: subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_FAIL_MAIL_MESSAGE`
: message of such e-mail, by default it is:

```
Project:      $EVL_PROJECT
Workflow:     $EVL_WORKFLOW
Task:         $EVL_TASK
Order Date:   $EVL_ODATE
Sent to:      $EVL_MAIL_TO
Task log:     $EVL_TASK_LOG
Tail of log: $(tail $EVL_TASK_LOG)
```

where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_FAIL_SNMP=0`
: whether or not to send SNMP trap when the task fails.

`EVL_RUN_FAIL_SNMP_MESSAGE='$EVL_PROJECT FAILED'`
: SNMP message to be send.

Control retries:

`EVL_RUN_RETRY=0`
: the number of times it retries to run the task again. Zero means no retry and fail '`Run`' command once the given task fails.

`EVL_RUN_RETRY_INTERVAL=5m`
: the amount of time between retries. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Control parallel runs:

`EVL_RUN_MAX_PARALLEL=16`
: how many tasks in the workflow can run at once in parallel in one stage (i.e. before next '`Wait`')

`EVL_RUN_MAX_PARALLEL_CHECK_SEC=10`
: how many seconds to wait between checking maximum parallel runs

Control waiting:

`EVL_RUN_DEPENDENCIES_CHECK_SEC=10`
: how many seconds to wait between checking dependencies from dependencies file.

`EVL_RUN_TIME=24h`
: maximal run time, so if the task invoked by '`Run`' command is not finished after this amount of time, it is killed. The time is counted since the task is really running, not since the invocation (i.e. waiting time is not included). It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_FILE_INTERVAL=5m`
: the time interval between each check for a file(mask) existence. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_FILE_TIME=10h`
: maximal amount of time to wait for a file(mask). It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK=1`
: whether or not to wait for a lock file, i.e. if somebody is running the same task at the moment.

`EVL_RUN_WAIT_FOR_LOCK_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK_TIME=10h`
: maximal amount of time to wait for a lock file. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE=0`
: whether or not to automatically wait for previous Order Date of given task. Setting to 1 might be useful when you must run daily processing strictly in right order.

`EVL_RUN_WAIT_FOR_PREV_ODATE_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE_TIME=10h`
: maximal amount of time to wait for previous Order Date. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Warning control:

`EVL_RUN_WARN_MAIL=0`
: whether or not to send an email when there is warning

`EVL_RUN_WARN_MAIL_SUBJECT='$EVL_PROJECT WARNING'`
: subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_WARN_MAIL_MESSAGE`
: message of such e-mail, by default it is:

```
Project:      $EVL_PROJECT
Workflow:     $EVL_WORKFLOW
Task:         $EVL_TASK
Order Date:   $EVL_ODATE
Sent to:      $EVL_MAIL_TO
Task log:     $EVL_TASK_LOG
Tail of log:  $(tail $EVL_TASK_LOG)
```

where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_WARN_SNMP=0`
: whether or not to send SNMP trap when there is a warning

`EVL_RUN_WARN_SNMP_MESSAGE='$EVL_PROJECT WARNING'`
: SNMP message to be send in such case

Kubernetes control:

`EVL_RUN_K8S_CONTAINER_IMAGE="evl-tool:latest"`
: an image to run the task on

`EVL_RUN_K8S_CONTAINER_LIMIT_CPU="8000m"`
: maximum CPUs available for the task, decimal number is allowed, or "millicpu" can be used. E.g. "0.5" and "500m" are the same and means half of the CPU.

`EVL_RUN_K8S_CONTAINER_LIMIT_MEMORY="4Gi"`
: maximum memory for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_LIMIT_STORAGE="40Gi"`
: maximum ephemeral storage for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_REQUEST_CPU="2000m"`
: minimum requested CPUs for the task, decimal number is allowed, or "millicpu" can be used. E.g. "2.2" and "2200m" are the same.

`EVL_RUN_K8S_CONTAINER_REQUEST_MEMORY="1Gi"`
: minimum requested memory for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_REQUEST_STORAGE="20Gi"`
: minimum ephemeral storage for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_NAMESPACE="default"`
: Kubernetes namespace under which tasks suppose to run

`EVL_RUN_K8S_PERSISTENT_BUCKET`
: mandatory variable with persistent (AWS S3) bucket, where: 1. the EVL license is stored. i.e. ‘`s3://<some_bucket>/evl_license_key`’ 2. logs are collected in ‘`evl-log`’ directory, i.e. ‘`s3://<some_bucket>/evl-log`’ 3. EVL projects whose tasks to be run, e.g. ‘`s3://<some_bucket>/<some_project>`’, (it copies the tasks’ definitions from this location). The value must be only a bucket name, without ‘`s3://`’.

`EVL_RUN_K8S_SERVICE_ACCOUNT_NAME="default"`
: Kubernetes Service Account Name

`EVL_RUN_K8S_SHM_LIMIT="1Gi"`
: Shared Memory Size Limit of the Kubernetes Pod. (Shared memory is used by shared lookups.)

`EVL_RUN_K8S_RETRY=0`
: the ‘`backoffLimit`’ parameter in Kubernetes job definition. This variable defines how many times to restart a Kubernetes job, this number of retires is on the Kubernetes level, so it is different from the variable ‘`EVL_RUN_RETRY`’, which is on the current machine level.

`EVL_RUN_K8S_TTL_AFTER_FINISHED=10`
: set spec.ttlSecondsAfterFinished in Kubernetes Job definition, non-zero value is good to obtain logs from finished (i.e. Compleded or Failed) tasks. (TTL means ’Time to Live’.)

SSH control:

TBA

Examples

Commandline invocation:

1. To restart a workflow from the beginning:
   ```
   evl run --restart workflow/load_invoices.ewf
   ```
2. Following command runs an EVL job with yesterday ODATE showing progress

   evl run job/aggreg_invoices.evl –odate=yesterday –progress –project=/full/path/to/project

   or when current directory is a project one:
   ```
   evl run job/aggreg_invoices.evl --odate=yesterday --progress
   ```

Within EWS file usage:

1. To run an EVL job in an EVL Workflow (i.e. within an ‘`ews`’ file), and try once more when job fails:
   ```
   Run 1r aggreg_invoices.evl
   ```

2. Following invocation means to run common_job.sh (fail if not finished within 2 hours) and then run '`job/stage.invoices.evl`', fail if (each run) does not finish within 4 hours, and try to restart two times when fail:

   ```
   Run 2h    job/common_job.sh  --project=/full/path/to/other/project \
       4h 2r job/stage.invoices.evl
   ```

13.17 Set *(since EVL 2.0)*

Manually set status to EVL task (i.e. job, workflow or script, or waiting for a file). Keep in mind, that this command is not intended for standard usage. It is only for special cases when something went wrong and you need to set status manually.

Possible states are:

'`reru`'
: set state to '`RERU`', i.e. '`To rerun`'.

'`wait`'
: set state to '`WAIT`', i.e. '`Waiting`'.

'`runn`'
: set state to '`RUNN`', i.e. '`Running`'.

'`fail`'
: set state to '`FAIL`', i.e. '`Failed`'.

'`canc`'
: set state to '`CANC`', i.e. '`Cancelled`'.

'`skip`'
: set state to '`SKIP`', i.e. '`Skipped`'.

'`succ`'
: set state to '`SUCC`', i.e. '`Successful`'.

'`arch`'
: set state to '`ARCH`', i.e. '`Archived`'.

'`dele`'
: set state to '`DELE`', i.e. '`Deleted`'.

Synopsis

```
evl set
  <state> ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project_dir>] [-v|--verbose]

evl set
  ( --help | --usage | --version )
```

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Mark '`some_job.evl`' as successfully finished with current date as '`$EVL_ODATE`':

```
evl set succ some_job.evl
```

13.18 Skip *(since EVL 2.0)*

Use '`skip`' to produce log entry like in the case an EVL task (i.e. job, workflow or script, or waiting for a file) would be successfully finished (in fact marked as '`SKIP`'). This is useful when other jobs or workflows are waiting for such EVL task.

It is actually an alias to '`evl set skip`' commands.

Synopsis

```
evl skip
  <state> ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project>] [-v|--verbose]

evl skip
  ( --help | --usage | --version )
```

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. Mark 'some_job.evl' as successfully finished with current date as Order Date:

   ```
   evl skip some_job.evl --odate today
   ```

13.19 Sleep *(since EVL 2.4)*

Use 'Sleep' to fire previously defined EVL tasks (i.e. jobs/workflow/scripts or waiting for a file) by 'Run' command. Then wait specified amount of `<time>`.

`<time>` can be specified in seconds, minutes, hours or days, so suffix 's', 'm', 'h' or 'd' need to be specified to the number. If no unit is specified, seconds are assumed.

Sleep is useful when need to run jobs/workflows in parallel, but shifted way. Or in the case you need to spread many short (but intensive) jobs in the time, so kind of serial, but let them overlap.

While a workflow is invoked by 'continue', then already fired Sleep commands are ignored. So restarting a workflow this way will not stuck on already finished Sleeps.

`Sleep`
: is to be used in EVL workflow structure definition file, i.e. in EWS file.

`evl sleep`
: is intended for standalone usage, i.e. to be invoked from command line.

EWS is EVL workflow definition file, for details see man evl-ews(5).

Synopsis

```
Sleep
  <time>[smhd]

evl sleep
  <time>[smhd] [-v|--verbose]

evl sleep
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To run a job every 15 minutes, ignore failures:

```
export EVL_RUN_FAIL=0

let i=0
while (( i < 1440 ))
do
  # Using printf because of proper sorting by file name
  Run flowmon_load.$(printf "%04d" $i).evl
  Sleep 15m
let i+=15
done

Wait
```

2. To fire 100 jobs in parallel, but shifted by 20 seconds:

```
for i {1..100}
do
  Run $i.evl
  Sleep 20
done
Wait
```

13.20 Spark *(since EVL 2.0)*

In case jar file is specified (i.e. file with mask *.jar), it invokes:

```
$EVL_SPARK_SUBMIT <spark_submit_options> <jar_file> --name <name>
```

where EVL_SPARK_SUBMIT is '`spark-submit`' by default.

When other than jar file is used, then it firstly build the code by '`$EVL_SPARK_BUILD`', which is by default '`sbt`', and then run such jar file in above manner.

Synopsis

```
Spark
  ( <jar_file> | <scala_source> ) [--name <name>]

evl spark
  ( <jar_file> | <scala_source> ) [--name <name>]
  [--verbose]

evl spark
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

Run already built scala code in YARN:

```
export EVL_SPARK_SUBMIT="--master yarn --executor-memory 2G
                         --conf spark.executor.memoryOverhead=4G"
Spark agregate_something.jar --name aggregate_something
```

Run scala code in YARN:

```
export EVL_SPARK_SUBMIT="--master yarn --executor-memory 2G
                         --conf spark.executor.memoryOverhead=4G"
Spark agregate_something.scala --name aggregate_something
```

13.21 Status *(since EVL 2.4)*

To get the latest state of an EVL task (i.e. job, workflow or script, or waiting for a file).

Synopsis

```
evl status
  ( <run_id> | <task_name> [-o|--odate=<odate>] )
  [-p|--project=<project>] [-v|--verbose]

evl status
  ( --help | --usage | --version )
```

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable ‘`EVL_ODATE`’ is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. To get the last state of ‘`some_job.evl`’ with today Order Date:
   ```
   evl status some_job.evl --odate today
   ```
2. To get the last state of task with Run ID 3222 of the given project:
   ```
   evl status 3222 -p /data/project/roaming
   ```

13.22 Test *(since EVL 2.1)*

On local file system works like standard GNU/Linux ‘`test`’ command. If `<path>` starts with ‘`hdfs://`’, then use function ‘`evl_hdfs_test`’, which is by default ‘`hadoop fs -test`’. If `<path>` starts with ‘`s3://`’, then use function ‘`evl_s3_test`’.

Synopsis

```
Test
  -[defsz] <path>

evl test
  ( --help | --usage | --version )
```

Options

`-d`

return 0 if <path> exists and is a directory

`-e`

return 0 if <path> exists

`-f`

return 0 if <path> exists and is a regular file

`-s`

return 0 if file <path> exists and has a size greater than zero

`-z`

return 0 if file <path> is zero bytes in size, else return 1

Examples

TBA

13.23 Touch *(since EVL 2.7)*

Update the access and modification times of each <file> to the current time. Each `<file>` argument that does not exist is created empty.

Each `<file>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme ‘`hdfs://`’ it calls function ‘`evl_hdfs_touch`’, which is by default ‘`hdfs dfs -touchz`’.

For scheme ‘`sftp://`’ it calls function ‘`evl_sftp_touch`’.

Synopsis

```
Touch
  <file>...
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Simple usage examples:

   ```
   Touch hdfs:///some/path/
   Touch /some/local/machine/path/
   ```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

   ```
   # on DEV:
   OUTPUT_DIR=/data/output
   # on PROD:
   OUTPUT_DIR=hdfs:///data/output
   ```

   and then in '`evs`' file:

   ```
   Touch "$OUTPUT_DIR"
   ```

13.24 Wait *(since EVL 2.1)*

It waits for successful finish of all previous components of an EVL job (in an EVS file) or all previous '`Run`' commands in an EVL workflow (in an EWS file). So it acts similar way as standard Bash '`wait`' command.

Once all previous components/parts of the job/workflow successfully finish the job/workflow continue further.

EVS is EVL job structure definition file, for details see '`man 5 evl-evs`'.

EWS is EVL workflow structure definition file, for details see '`man 5 evl-ews`'.

EVL job (EVS file):

If any component of an EVL job fails, it immediately cancel the whole job.

There is neither argument nor option for '`Wait`' command in EVL job, i.e. in EVS file. More precisely: all arguments are ignored, so it can be used for comments.

EVL workflow (EWS file):

If any invoked EVL job/workflow (by '`Run`' command) fails, then cancel only given '`Run`' command and continue in processing others. Once all '`Run`' commands finish (either successfully or fail), then '`Wait`' command fails unless '`EVL_WAIT_FAIL`' is set to 0.

When no `<job>`, `<workflow>` or `<script>` is specified, further processing will wait `<time>` for all previously fired parts (i.e. '`Run`' commands) and fail if at least one of them fails.

When no `<time>` is specified by '`--time`' option, then wait at most '`$EVL_WAIT_TIME`', which is by default 10 hours.

`<time>` can be secified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

When `<job>`, `<workflow>` or `<script>` is specified, it waits for successful run of such job/workflow/script of the current project and of the current `<odate>`. Different `<project>` and/or `<odate>` can be specified by the particular options.

Synopsis

```
Wait
  ( <job>.evl | <workflow>.ewf | <script>.sh )
  [-p|--project=<project>] [-o|--odate=<odate> | -y|--yesterday]
  [-t|--time=<time>]

Wait
  [-m|--myself [-o|--odate=<odate> | -y|--yesterday]]
  [-t|--time=<time>]

evl wait
  ( --help | --usage | --version )
```

Options

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Environment Variables

`EVL_WAIT_FAIL=1`
: EVL workflow only. Whether or not to fail the whole workflow when the ‘`Run`’ command fails, so when zero is set, the workflow continue regardless task failures

`EVL_WAIT_INTERVAL=2s`
: EVL workflow only. The time interval between each check for ‘`Wait`’ command. It can be specified in seconds, minutes, hours or days, so suffix ‘`s`’, ‘`m`’, ‘`h`’ or ‘`d`’ need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_WAIT_TIME=10h`
: EVL workflow only. Maximal amount of time to wait for a previous ‘`Run`’ commands to finish. It can be specified in seconds, minutes, hours or days, so suffix ‘`s`’, ‘`m`’, ‘`h`’ or ‘`d`’ need to be specified to the number. If no unit is specified, seconds are assumed.

Examples

EVL job

1. Run EVL job in two steps:

```
Read  file.json INPUT   evd/file.evd --text-input
Map   INPUT     MAPPED  evd/file.evd evd/stage.evd evm/file.evm
Write MAPPED    stage.parquet        evd/stage.evd
```

```
Wait "for creation of parquet file."

Run   "impala \"refresh table stage;\""

End
```

EVL workflow

2. Wait for myself (i.e. the same workflow yesterday) to finish:
   ```
   Wait --myself
   ```
   or the inother words:
   ```
   Wait --myself --yesterday
   ```
   or shortly also:
   ```
   Wait -my
   ```
3. Wait for all jobs to finish, but at most 1 hour, then fail:
   ```
   Run file2stage.bills.evl update.bills.evl
   Run file2stage.invoices.evl
   Wait --time 1h
   ```
4. Wait (forever) for successful run of the job '`file2stage_example.evl`', then continue:
   ```
   Wait job/file2stage.example.evl
   ```
5. Wait (at most 1 day) for the job '`export_job.evl`' before continue:
   ```
   Wait job/export_job.evl --time 1d
   ```
6. Wait (at most 120 minutes) for the job '`sftp_billing.evl`' of the different project '`billing`' (of current ODATE):
   ```
   Wait job/sftp_billing.evl --project billing --time 120m
   ```
7. Wait (at most 300 seconds) for the job '`sftp_billing.evl`' of the different project '`billing`' of the 20260121}:
   ```
   Wait job/sftp_billing.evl --project billing --odate "20260121" --time 300
   ```
8. Wait (at most 6 hours) for previous run of given workflow. (Variable '`ODATE_MINUS1`' has to be set by you.):
   ```
   Wait other_workflow.ewf --odate $ODATE_MINUS1 --time 6h
   ```

14 EVM Mappings

Important: Any C++ functions can be used in EVL mapping. Many of the following EVL functions only helps handling '`nullptr`', which represents NULL values.

All the functions are further sorted by name, so for better orientation here is an overview by usage groups.

Mapping Functions

- Section 14.1 [Output Functions], page 174, to work with mapping's outputs, rejects, etc.
- Section 14.2 [String Functions], page 176, to manipulate strings
- Section 14.3 [Date and Time Functions], page 184, to manipulate dates and times
- Section 14.4 [Randomization Functions], page 184, to generate random values
- Section 14.5 [Anonymization Functions], page 185, to anonymize in various ways
- Section 14.6 [Encryption Functions], page 188, encryption, decryption
- Section 14.7 [Conversion Functions], page 188, data types conversions
- Section 14.8 [IP Addresses Functions], page 189, to manipulate IPv4 or IPv6 addresses
- Section 14.9 [Logical Functions], page 190, functions which returns true or false
- Section 14.10 [Checksum Functions], page 192, to do checksums
- Section 14.11 [Mathematical Functions], page 192, mathematical calculations
- Section 14.12 [Lookup Functions], page 193, lookup functions
- Section 14.13 [Other Functions], page 196, e.g. '`first_not_null()`'

Important: There are several special characters in an EVD field name which must be handled different way in EVM mapping:

Number at the beginning
: When the field name starts with a number then in the mapping must be used prefixed by underscore. E.g. field `01_bill_type` would be referenced in mapping as `_01_bill_type`.

Non-alphanumeric characters
: All non-alphanumeric characters in field name have to be referenced in mapping as underscore. E.g. field `$bill type (9)` would be referenced in mapping as `_bill_type__9_`.

14.1 Output Functions

There are several functions which can modify standard component behaviour regarding output of each record.

14.1.1 discard and reject

Using '`discard()`' simply doesn't output current record anywhere. (It has better performance than output to `/dev/null`, i.e. using `Trash`.)

But then it doesn't end processing the mapping, so mostly the use would be:

```
discard(); return;
```

which immediately ends up the current mapping and iterate to another record.

Compare to ‘discard()’, ‘reject()’ function redirects input (with input evd) into specified output following way:

```
reject();                   // redirect input record to reject port
reject(6);                  // redirect input record to output /dev/fd/6
reject("out.csv");          // redirect input record to file "out.csv",
                            // if such exists, will be overwritten
reject("out.csv", open_mode::overwrite); // same as previous
reject("out.csv", open_mode::append);    // same as previous,
                                         // but append, not overwrite
reject("out.csv", open_mode::create);    // redirect input to "out.csv",
                                         // but fail if such exists
```

Function headers:

```
void discard() const;

void reject() const;
void reject(const int file_descriptor) const;
void reject(const char* const path, \
            const open_mode mode = open_mode::overwrite);
void reject(const std::string& path, \
            const open_mode mode = open_mode::overwrite);
```

And variants for join mapping:

```
void reject_left(const char* const path, \
                 const open_mode mode = open_mode::overwrite);
void reject_right(const char* const path, \
                  const open_mode mode = open_mode::overwrite);
void reject_left(const std::string& path, \
                 const open_mode mode = open_mode::overwrite);
void reject_right(const std::string& path, \
                  const open_mode mode = open_mode::overwrite);
```

14.1.2 add_record and output

Both these functions produce records with the output evd. Example first:

```
add_record();               // add new record to stdout
add_record(4);              // add new record to output /dev/fd/4
add_record("out.csv");      // add new record to file "out.csv",
                            // if such exists, will be overwritten
add_record("out.csv", open_mode::overwrite); // same as previous
add_record("out.csv", open_mode::append);    // same as previous,
                                             // but append, not overwrite
add_record("out.csv", open_mode::create);    // add record to "out.csv",
                                             // but fail if such exists
```

Function ‘add_record()’ sends current output record to the specified output and continue in processing of the mapping.

Function ‘output()’ behave the same way as ‘add_record()’, but doesn’t produce additional record, only redirect the current output record.

The ‘open_mode’ is taken only from the very first call of the function, the others are ignored and the file is still open for writing in the same mode. So for example calling in some mapping

```
output("out.csv")
```

will delete file `out.csv` if it exists and starts to append every output record, keeping this file open.

Functions headers:

```
void add_record() const;
void add_record(const int file_descriptor) const;
void add_record(const char* const path, \
                const open_mode mode = open_mode::overwrite) const;
void add_record(const std::string& path, \
                const open_mode mode = open_mode::overwrite) const;

void output(const int file_descriptor = 4);
void output(const char* const path, \
            const open_mode mode = open_mode::overwrite);
void output(const std::string& path, \
            const open_mode mode = open_mode::overwrite);
```

Note: '`output(-1)`' is the same as '`discard()`'.

14.1.3 unmatched_left, unmatched_right

In `Join` component there these two functions which catch unmatched left and/or right join.

Usual example is for `Join` of `--type left`:

```
out->field = left->field;
if (!right) unmatched_left();
```

It means, that to the output goes inner joined records and to the `--left-unmatched` port goes not-joined records from left.

Function headers:

```
void reject_left(const int output = 7);
void reject_right(const int output = 7);
```

14.1.4 reject_left, reject_right

Actually similar to '`unmatched_left/right()`' functions, just redirect input record (with input `evd`).

Function headers:

```
void reject_left(const int output = 7);
void reject_right(const int output = 7);
```

14.1.5 warn and fail

Function '`warn()`' add a warning message to the standard error of the job, but the mapping continue processing the input records. To terminate the mapping and let the job fail, use function '`fail()`'.

```
if (!in->name) fail("Name is missing and is mandatory.");
if (!in->email) warn("Careful, e-mail is missing for: " + *in->name);
```

14.2 String Functions

All string manipulation 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:

```
out->field  = str_function(in->field);
*out->field = str_function(*in->field);
```

are basically the same, but the second one will fail in case ‘`in->field`’ will be NULL (i.e. ‘`nullptr`’).

There are these two rules in all 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.

14.2.1 length *(since EVL 2.0)*

Returns the length of given string.

For ‘`nullptr`’ it returns again ‘`nullptr`’.

Example:

```
length((string)"Some text")     // return 9
length(nullptr)                 // return nullptr
```

In mapping it might look like this (without pointers):

```
out->str_len = length(in->first_name);
```

14.2.2 split *(since EVL 1.3)*

Example:

```
split("Some text, another text.", ' ')
     // returns vector ["Some", "text,", ‘another’, "text."]
```

When the first argument is ‘`nullptr`’, it returns ‘`nullptr`’.

In mapping it might look like this (without pointers):

```
static std::vector<std::string> name_vec;

name_vec = split(*in->full_name", ' ');
*out->first_name = name_vec[0];
*out->last_name  = name_vec[1];
```

or (preferably) using pointers:

```
static std::vector<std::string*>* name_vec;

name_vec = split(in->full_name", ' ');
out->first_name = name_vec[0];
out->last_name  = name_vec[1];
```

Function headers:

```
std::vector<std::string>   split(const std::string& str, \
                                 const char delimiter);
std::vector<std::string*>* split(const std::string* const str, \
                                 const char delimiter);
```

14.2.3 starts_with, ends_with *(since EVL 2.0)*

True if a string starts or ends with the given substring.

When the first argument is 'nullptr', it returns False.

Example:

```
starts_with("Some text", "Some")   // return True
starts_with("Some text", "x")      // return False
starts_with(nullptr, "x")          // return False
ends_with("Some text", "ext")      // return True
ends_with("Some text", "x")        // return False
```

In mapping it might look like this:

```
*out->test_field = starts_with(in->test_field ? "OK" : "NOK" ;
```

Function headers:

```
bool starts_with(const std::string& str, const char* const prefix);
bool starts_with(const std::string* const str, const char* const prefix);
bool starts_with(const std::string& str, const std::string& prefix);
bool starts_with(const std::string* const str, const std::string& prefix);
```

```
bool ends_with(const std::string& str, const char* const suffix);
bool ends_with(const std::string* const str, const char* const suffix);
bool ends_with(const std::string& str, const std::string& suffix);
bool ends_with(const std::string* const str, const std::string& suffix);
```

14.2.4 str_compress, str_uncompress *(since EVL 2.0)*

Compress/uncompress the given string. Examples which return pointers:

```
str_compress(in->string_field_to_compress)      // snappy by default
str_compress(in->string_field_to_compress, compression::gzip)
str_compress(in->snappy_field)                  // snappy by default
str_compress(in->gzipped_field, compression::gzip)
```

Examples which return string values:

```
str_compress(*in->string_field_to_compress)     // snappy by default
str_compress(*in->string_field_to_compress, compression::gzip)
str_compress(*in->snappy_field)                 // snappy by default
str_compress(*in->gzipped_field, compression::gzip)
```

When the first argument is 'nullptr', it returns 'nullptr'.

In mapping it might look like this:

```
out->gzipped_field = str_compress(in->string_field);
```

Function headers:

```
std::string str_compress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_compress(const std::string* const str, \
             const compression method = compression::snappy);
```

```
std::string str_uncompress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_uncompress(const std::string* const str, \
             const compression method = compression::snappy);
```

14.2.5 str_count *(since EVL 1.3)*

It counts the number of occurrences of given string or character. Example:

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

When the first argument is 'nullptr', it returns 'nullptr'.

In mapping it might look like this (using pointers):

```
out->jan_cnt  = str_count(in->first_name", "Jan");
```

or without pointers:

```
*out->jan_cnt = str_count(*in->first_name", "Jan");
```

Function headers:

```
std::size_t  str_count(const std::string& str, const char ch);
std::size_t* str_count(const std::string* const str, const char ch);
std::size_t  str_count(const std::string& str, const char* const substr);
std::size_t* str_count(const std::string* const str, \
                       const char* const substr);
std::size_t  str_count(const std::string& str, const std::string& substr);
std::size_t* str_count(const std::string* const str, \
                       const std::string& substr);
```

14.2.6 str_index, str_rindex *(since EVL 2.0)*

`str_index(str,substr)`
: it returns the index (counted from 0) of the first occurrence of the given substring,

`str_rindex(str,substr)`
: it returns the index (counted from 0) of the last occurrence of the given substring.

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

When the string is '`nullptr`', it returns '`nullptr`'.

Examples:

```
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
```

Function headers:

```
std::int64_t  str_index(const std::string& str, const char* const substr);
std::int64_t* str_index(const std::string* const str, \
                        const char* const substr);
std::int64_t  str_index(const std::string& str, const std::string& substr);
std::int64_t* str_index(const std::string* const str, \
                        const std::string& substr);

std::int64_t  str_rindex(const std::string& str, const char* const substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const char* const substr);
std::int64_t  str_rindex(const std::string& str, const std::string& substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const std::string& substr);
```

14.2.7 str_join *(since EVL 2.4)*

`str_join(vector_of_strings,delimiter)`
it returns the string of concatenated vector members, delimited by a specified delimiter.

When the vector is '`nullptr`', it returns '`nullptr`'.

Examples of a mapping:

```
static std::vector<std::string> x{"Here", "is", "a", "hardcoded", "vector."};

*out->x_spaced = str_join(x,' ')   // return "Here is a hardcoded vector."
*out->x_dashed = str_join(x,'-')   // return "Here-is-a-hardcoded-vector."
*out->x_longer = str_join(x,"---") // return "Here---is---a---hardcoded---vector."
```

Function headers:

```
std::string  str_join(const std::vector<std::string>& strings, \
                      const char delimiter);
std::string* str_join(const std::vector<std::string*>* strings, \
                      const char delimiter);
std::string  str_join(const std::vector<std::string>& strings, \
                      const std::string_view delimiter);
std::string* str_join(const std::vector<std::string*>* strings, \
                      const std::string_view delimiter);
```

14.2.8 str_mask_left, str_mask_right *(since EVL 2.1)*

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 '`nullptr`', these functions return '`nullptr`'.

Function headers:

```
std::string  str_mask_left(const std::string& str, \
                  const std::size_t keep, const char ch = '*');
std::string* str_mask_left(const std::string* const str, \
                  const std::size_t keep, const char ch = '*');
std::string  str_mask_right(const std::string& str, \
                  const std::size_t keep, const char ch = '*');
std::string* str_mask_right(const std::string* const str, \
                  const std::size_t keep, const char ch = '*');
```

14.2.9 str_pad_left, str_pad_right *(since EVL 2.1)*

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 '`nullptr`', these functions return '`nullptr`'.

Function headers:

```
std::string  str_pad_left(const std::string& str, \
                 const std::size_t length, const char ch = ' ');
std::string* str_pad_left(const std::string* const str, \
                 const std::size_t length, const char ch = ' ');
std::string  str_pad_right(const std::string& str, \
                 const std::size_t length, const char ch = ' ');
std::string* str_pad_right(const std::string* const str, \
                 const std::size_t length, const char ch = ' ');
```

14.2.10 str_replace *(since EVL 1.3)*

Examples:

```
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 '`nullptr`', it returns '`nullptr`'.

In mapping it might look like this:

```
out->name = str_replace(in->name", ' ', '-');
```

Function headers:

```
std::string  str_replace(const std::string& str, \
                 const char old_ch, const char new_ch);
std::string* str_replace(const std::string* const str, \
                 const char old_ch, const char new_ch);
std::string  str_replace(const std::string& str, \
                 const char* const old_substr, const char* const new_substr);
std::string* str_replace(const std::string* const str, \
                 const char* const old_substr, const char* const new_substr);
std::string  str_replace(const std::string& str, \
                 const std::string& old_substr, const std::string& new_substr);
std::string* str_replace(const std::string* const str, \
                 const std::string& old_substr, const std::string& new_substr);
```

14.2.11 str_index, str_rindex *(since EVL 2.0)*

`str_index(str,substr)`
: it returns the index (counted from 0) of the first occurrence of the given substring,

`str_rindex(str,substr)`
: it returns the index (counted from 0) of the last occurrence of the given substring.

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

When the string is '`nullptr`', it returns '`nullptr`'.

Examples:

```
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
```

Function headers:

```
std::int64_t  str_index(const std::string& str, const char* const substr);
std::int64_t* str_index(const std::string* const str, \
                        const char* const substr);
std::int64_t  str_index(const std::string& str, const std::string& substr);
std::int64_t* str_index(const std::string* const str, \
                        const std::string& substr);
std::int64_t  str_rindex(const std::string& str, const char* const substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const char* const substr);
std::int64_t  str_rindex(const std::string& str, const std::string& substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const std::string& substr);
```

14.2.12 str_to_base64, base64_to_str *(since EVL 2.6)*

Encode/decode string to/from Base64 form.

When the first argument is 'nullptr', it returns also 'nullptr'.

Examples:

```
  str_to_base64("Some\r\nbíňářý text.")   // return "U29tZQ0KYsOtxYjDocWZw70gdGV4dC4="
  base64_to_str("U29tZQ0KYsOtxYjDocWZw70gdGV4dC4=")   // return "Some\r\nbíňářý text."
```

Function headers:

```
std::string  str_to_base64(const std::string& str);
std::string* str_to_base64(const std::string* const str);
std::string  base64_to_str(const std::string& str);
std::string* base64_to_str(const std::string* const str);
```

14.2.13 str_to_hex, hex_to_str *(since EVL 2.0)*

Convert string or ustring to its hexadecimal representation and vice versa. (Ustring support has been added in EVL v2.6.)

When the first argument is 'nullptr', it returns also 'nullptr'.

Examples:

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

Function headers:

```
std::string  str_to_hex(const std::string& str);
std::string* str_to_hex(const std::string* const str);
ustring      str_to_hex(const __detail::u16str& str);
ustring*     str_to_hex(const ustring* const str);
std::string  hex_to_str(const std::string& str);
std::string* hex_to_str(const std::string* const str);
ustring      hex_to_str(const __detail::u16str& str);
ustring*     hex_to_str(const ustring* const str);
```

14.2.14 str_compress, str_uncompress *(since EVL 2.0)*

Compress/uncompress the given string. Examples which return pointers:

```
  str_compress(in->string_field_to_compress)       // snappy by default
```

```
str_compress(in->string_field_to_compress, compression::gzip)
str_compress(in->snappy_field)                  // snappy by default
str_compress(in->gzipped_field, compression::gzip)
```

Examples which return string values:

```
str_compress(*in->string_field_to_compress)     // snappy by default
str_compress(*in->string_field_to_compress, compression::gzip)
str_compress(*in->snappy_field)                 // snappy by default
str_compress(*in->gzipped_field, compression::gzip)
```

When the first argument is '`nullptr`', it returns '`nullptr`'.

In mapping it might look like this:

```
out->gzipped_field = str_compress(in->string_field);
```

Function headers:

```
std::string str_compress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_compress(const std::string* const str, \
             const compression method = compression::snappy);
std::string str_uncompress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_uncompress(const std::string* const str, \
             const compression method = compression::snappy);
```

14.2.15 substr *(since EVL 2.0)*

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 '`nullptr`', function returns '`nullptr`'.

Function headers:

```
std::string  substr(const std::string& str, const std::size_t pos = 0,
      const std::int64_t count = std::numeric_limits<std::int64_t>::max());
std::string* substr(const std::string* const str, const std::size_t pos = 0,
      const std::int64_t count = std::numeric_limits<std::int64_t>::max());
```

14.2.16 trim, trim_left, trim_right *(since EVL 1.0)*

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.

When the first argument is '`nullptr`', these functions return '`nullptr`'.

Function headers:

```
std::string  trim(const std::string& str, const char ch = ' ');
std::string* trim(const std::string* const str, const char ch = ' ');
```

```
std::string  trim_left(const std::string& str, const char ch = ' ');
std::string* trim_left(const std::string* const str, const char ch = ' ');

std::string  trim_right(const std::string& str, const char ch = ' ');
std::string* trim_right(const std::string* const str, const char ch = ' ');
```

14.2.17 uppercase, lowercase *(since EVL 1.0)*

Examples:

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

When the argument is '`nullptr`', these functions return '`nullptr`'.

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

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

```
static std::locale de_locale("de_DE.utf8");
*out->field_upcase = uppercase(*in->field, de_locale);
```

It is possible to specify the locale in the function as string, but using the static specification of locale is recommended due to performance.

Function headers:

```
std::string  uppercase(const std::string& str);
std::string* uppercase(const std::string* const str);
std::string  uppercase(const std::string& str, const std::locale& locale);
std::string* uppercase(const std::string* const str, const std::locale& locale);

std::string  lowercase(const std::string& str);
std::string* lowercase(const std::string* const str);
std::string  lowercase(const std::string& str, const std::locale& locale);
std::string* lowercase(const std::string* const str, const std::locale& locale);
```

14.3 Date and Time Functions

get_millisecond(timestamp*) get_microsecond(timestamp*) get_nanosecond(timestamp*) timezone_shift()

Difference:

```
auto diff = dt - datetime(2018,5,31,19,36,57); // 61 (seconds)
auto diff =  d - date("2017-04-02");           // -6 (days)
```

Let's summarize the logic:

```
date - int  => date          datetime - int      => datetime
date - date => int           datetime - datetime => int
```

14.4 Randomization Functions

For randomization functions are used same rules regarding '`nullptr`' as for string functions.

`randomize()` *(since EVL 2.1)*

Examples:

```
// random int from whole int range
out->random_int       = randomize(in->value);
// random int from interval < value - 1000 , value + 2000 >
out->random_int_range = randomize(in->value,-1000,2000);
```

`random_int()`
`random_long()`
`random_short()`
`random_char()` *(since EVL 2.1)*

Examples:

```
// random value from whole int range
out->random_value = random_int();
// random value from interval <1000,2000>
out->random_range = random_int(1000,2000);
```

`random_float()`
`random_double()` *(since EVL 2.1)*

Examples:

```
// random value from whole float range
out->random_value = random_float();
// random float value from interval <1000,2000>
out->random_range = random_float(1000,2000);
```

`random_decimal()` *(since EVL 2.1)*

Examples:

```
// random value from whole decimal range
out->random_value = random_decimal();
// random float value from interval <1000,2000>
out->random_range = random_decimal(1000,2000);
```

`random_date()`
`random_datetime()`
`random_timestamp()` *(since EVL 2.1)*

Examples:

```
// random date between 1970-01-01 and 2069-12-31
out->random_value = random_date();
// random date from this century
out->random_range = random_date(date("2000-01-01"), date("2099-12-31"));
```

`random_string()` *(since EVL 2.1)*

Examples:

```
// random string of length between 0 and 10
out->random_value = random_string();
// random string of length 5
out->random_range = random_string(5,5);
```

14.5 Anonymization Functions

For all anonymization functions there are again the same rules as for string functions, i.e.:

- when the argument is '`nullptr`', it returns again '`nullptr`';
- when the (first) argument is '`pointer`', it returns again '`pointer`'.

14.5.1 anonymize

`anonymize(str, keep_chars, keep_char_class = false)`
`anonymize(str, min_length, max_length)` *(since EVL 2.1)*

First argument '`str`' is mandatory and is of data type '`string`' or '`ustring`'. The function then returns such data type as well.

Parameter '`keep_chars`' is a string of characters which should be kept as is, i.e. such characters are not anonymized. Mostly it makes sense to use a space here, but for example to anonymize an email you can specify `"@."`. For '`ustring`' input it must be '`ustring`' as well, so for an email example `u"@."`

When parameter '`keep_chars_class`' is '`true`', then capital letters will be again capitals, lowercase letters stay lowercased and numbers will be numbers again.

Arguments '`min_length, max_length`' says how long the result could be. When no '`min_length, max_length`' parameters are used, then it returns a string or ustring of the same length as input.

Mapping examples:

```
out->anonymized_name = anonymize(in->name);
 // "Mircea Eliade" -> "icDoudVhaXYll" (same length)

out->anonymized_name = anonymize(in->name, " ");
 // "Mircea Eliade" -> "kJsqzt ZhGFts" (keep space)

out->anonymized_name = anonymize(in->name, " Maeiou");
 // "Mircea Eliade" -> "Misqea Jhiade" (keep also letters M,a,e,i,o,u)

out->anonymized_name = anonymize(in->name, " ", true);
 // "5 Mircea Eliade" -> "9 Piosdf Kiudpp" (keep space and char class)

out->anonymized_name = anonymize(in->name, 2, 10);
 // "Mircea Eliade" -> "jTro"       (length between 2 and 10)
 // "Franz Kafka"   -> "ksgTzDhoQf" (length between 2 and 10)

out->anonymized_name = anonymize(in->name, 0, length(in->name));
 // "Mircea Eliade" -> "lkdUuZytSd"
 // "Franz Kafka"   -> ""  // might be a NULL if 'name' is nullable
```

`anonymize(ustr, locale, keep_chars, keep_char_class = false)`
`anonymize(ustr, locale, min_length, max_length)` *(since EVL 2.5)*

First argument '`ustr`' is mandatory and is of data type '`ustring`'. The function returns such data type as well.

Arguments '`keep_chars`', '`keep_chars_class`' and '`min_length, max_length`' are the same as for previous variant of the function. Just '`keep_chars`' must be of ustring data type here.

Parameter '`locale`' is an instance of class ulocale defined in mapping, so for example the following mapping will produce anonymized (ustring) output consists of Spanish letters.

```
static ulocale my_locale("es_ES");
out->text_field =
    anonymize(u"Some text in Spanish.", my_locale, 1, 10);
```

Mapping examples with `name` and `anonymized_name` as `ustring` data type:

```
out->anonymized_name = anonymize(in->name);
 // "Leoš Janáček" -> "fQlKUHlduGus" (same length)

out->anonymized_name = anonymize(in->name, u" ");
 // "Leoš Janáček" -> "hGrT iUjSFeQ" (keep space)

out->anonymized_name = anonymize(in->name, u" š");
```

```
 // "Leoš Janáček" -> "jTDš oIZqqWv" (keep also letter š)

out->anonymized_name = anonymize(in->name, u" aeiou", true);
 // "8 Leoš Janáček" -> "3 Peoi Kařawec" (keep vowels and char class)

out->anonymized_name = anonymize(in->name, 2, 10);
 // "Bedřich Smetana" -> "SwpAq" (length between 2 and 10)
 // "Antonín Dvořák"  -> "Qs"    (length between 2 and 10)

out->anonymized_name = anonymize(in->name, 0, length(in->name));
 // "Bedřich Smetana" -> "HsgIusTFErq"
 // "Antonín Dvořák"  -> "" // might be a NULL if 'name' is nullable
```

`anonymize(number, min, max)` *(since EVL 2.1)*

To be used for '`number`' of all integral data types, for decimals and for floats. The function returns such data type then. Example (for :

```
anonymize((int)100, -5, 10);
 // return integer between 95 and 110 (incl.)
anonymize(  100.00, -5, 10);
 // return float   between 95 and 110 (incl.)
```

14.5.2 anonymize_uniq

`anonymize_uniq()` *(since EVL 2.1)*

Example:

```
out->anonymized_username = anonymize_uniq(in->id);
```

14.5.3 anonymize_iban

`anonymize_iban()` *(since EVL 2.4)*

Example:

```
string iban  = "NL91 ABNA 0417 1643 00"
string iban2 = "NL91ABNA0417164300"

anonymize_iban(iban)
            // return .... .... .... ....
anonymize_iban(iban2)
            // return ..................
anonymize_iban(iban, iban_anon::keep_country)
            // return NL.. .... .... .... ..
anonymize_iban(iban, iban_anon::keep_country_and_bank)
            // return NL.. ABNA .... .... ..
anonymize_iban(iban, iban_anon::whole, iban_form::grouped)
            // return .... .... .... .... ..
anonymize_iban(iban, iban_anon::whole, iban_form::compact)
            // return ..................
anonymize_iban(iban, iban_anon::keep_country, iban_form::compact)
            // return NL................
```

14.6 Encryption Functions

For all encryption functions there are again the same rules as for string functions, i.e.:

- when the argument is '`nullptr`', it returns again '`nullptr`';
- when the (first) argument is '`pointer`', it returns again '`pointer`'.

`rsa_encrypt_string(str, public_key)`
`rsa_encrypt_ustring(ustr, public_key)` *(since EVL 2.6)*

First argument '`(u)str`' is mandatory and is of data type '`string`' or '`ustring`'. In both cases the function returns '`string`' data type.

Second argument is also mandatory and contains the public key previously defined in the mapping by

```
static rsa_public_key public_key("/path/to/key.pub");
```

Encrypted string is actually binary data, so if there is a need to store this encrypted data in text mode, e.g. in CSV file, then for example '`str_to_base64`' function can be used. See example below.

`rsa_decrypt_string(str, private_key)`
`rsa_decrypt_ustring(str, private_key)` *(since EVL 2.6)*

First argument '`str`' is mandatory and is a (binary) string previously encrypted by '`rsa_encrypt_string`' or '`rsa_encrypt_ustring`'. It is necessary to keep the string or ustring couple, e.g. '`rsa_decrypt_ustring`' use for string encrypted by '`rsa_encrypt_ustring`'. In both case the function returns '`string`' data type.

Second argument is mandatory and contains the private key previously defined in the mapping by

```
static rsa_private_key private_key("/path/to/key.priv");
```

Mapping example for encryption

Both output fields are of type string, input field '`name`' is of type ustring.

```
static rsa_public_key pubkey("/path/to/key.pub");

out->name_encrypted_binary
          = rsa_encrypt_ustring(in->name,pubkey);
out->name_encrypted_textual
          = str_to_base64(rsa_encrypt_ustring(in->name,pubkey));

// Proper way in this particular example would be of course to use
// out->name_encrypted_textual
          = str_to_base64(out->name_encrypted_binary);
// to avoid applying encryption function twice
```

Mapping example for decryption afterwards

Both output fields are of type ustring and store the same value.

```
static rsa_private_key privkey("/path/to/key.priv");

out->name1 = rsa_decrypt_ustring(in->name_encrypted_binary,privkey);
out->name2 = rsa_decrypt_ustring(
                 base64_to_str(in->name_encrypted_textual),privkey);
```

14.7 Conversion Functions

14.7.1 to_<type> *(since EVL 1.0)*

`to_char(value)`
`to_uchar(value)`
`to_short(value)`
`to_ushort(value)`
`to_int(value)`
`to_uint(value)`
`to_long(value)`
`to_ulong(value)`
: return value of any (reasonable) data type converted to given integral data type.

`to_float(value)`
`to_double(value)`
: return value of any (reasonable) data type converted to float or double,

`to_decimal(value,n)`
: return value of any (reasonable) data type converted to decimal with scale '`n`' (i.e. decimal places).

`to_date(value)`
`to_time(value)`
`to_time_ns(value)`
`to_interval(value)`
`to_interval_ns(value)`
`to_datetime(value)`
`to_timestamp(value)`
: return value of any (reasonable) data type converted to given date/time data type.

`str_to_ipv4(str)`
`str_to_ipv6(str)`
: *(since EVL 2.4)*
convert string '`str`' to IPv4 or IPv6.

14.8 IP Addresses Functions

Typical IPv4 manipulation usage within a mapping:

```
// convert and assign IPv4 string into unsigned integer
out->ipv4_uint = str_to_ipv4(in->ipv4_string);
// or the other way
out->ipv4_string = ipv4_to_str(in->ipv4_uint);
```

Typical IPv6 manipulation usage within a mapping:

```
// suppose in->ipv6_string = "4567::123"
out->ipv6_normalized = ipv6_normalize(in->ipv6_string);
    // return "4567:0000:0000:0000:0000:0000:0000:0123"

// suppose in->ipv6_string = "0000:0000:0000:0004:5678:9098:0000:0654"
out->ipv6_compressed = ipv6_compress(in->ipv6_string);
    // return "::4:5678:9098:0000:654"
```

Or one can distinguish both IP versions:

```
if ( is_valid_ipv4(in->ip_string) ) {
  // act on IPv4
}
```

```
else if ( is_valid_ipv6(in->ip_string) ) {
  // act on IPv6
}
else {
  // act when neither is valid
}
```

There are these two rules in all IP 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.

14.8.1 IPv4 Functions *(since EVL 2.4)*

'`ipv4addr`'
: constructor

'`str_to_ipv4()`'
: convert string to uint32,

'`ipv4_to_str()`'
: convert uint32 to ipv4 string,

'`is_valid_ipv4()`'
: to check whether the string is valid IPv4.

14.8.2 IPv6 Functions *(since EVL 2.4)*

'`str_to_ipv6()`'
: convert string to uint128,

'`ipv6_to_str()`'
: convert uint128 to ipv6 string,

'`is_valid_ipv6()`'
: to check whether the string is valid IPv6,

'`ipv6_normalize()`'
: convert string to normalized IPv6 string,

'`ipv6_compress()`'
: convert string to compressed IPv6 string,

Examples

To get normalized and compressed IPv6:

```
// suppose in->ipv6_string = "0000:0000:22::0003:4"
out->ipv6_normalized = ipv6_normalize(in->ipv6_string);
    // "0000:0000:0022:0000:0000:0000:0003:0004"
out->ipv6_compressed = ipv6_compress(in->ipv6_string);
    // "0:0:22::3:4"
```

14.9 Logical Functions

14.9.1 is_equal *(since EVL 2.7)*

`is_equal(value1,value2)`
: return TRUE if '`value1`' is equal to '`value2`' or if both are null, otherwise it is FALSE. '`value`'s might be also pointers. For example following example is applicable:

```
is_equal(in->value_field1, in->value_field2)
```

14.9.2 is_in *(since EVL 2.4)*

`is_in(value, compare1, compare2, ...)`
: return TRUE if '`value`' is equal '`compare1`' or equal to '`compare2`', etc., otherwise it is FALSE. '`value`' doesn't need to be the same data type as compared values, but must be comparable. '`value`' and also compared list of values might be also pointers. For example following example is applicable:

```
is_in(in->some_uint, 123, in->some_long, 12.00, nullptr)
```

`is_in(value, vector)`
: return TRUE if '`value`' is equal at least one of the '`vector`' elements, otherwise it is FALSE.

14.9.3 is_valid_<type> *(since EVL 1.0)*

`is_valid_char(str)`
`is_valid_uchar(str)`
`is_valid_short(str)`
`is_valid_ushort(str)`
`is_valid_int(str)`
`is_valid_uint(str)`
`is_valid_long(str)`
`is_valid_ulong(str)`
: to check if given string '`str`' is valid integral data type,

`is_valid_float(str)`
`is_valid_double(str)`
: to check if given string '`str`' is valid float or double,

`is_valid_decimal(str,m,n,dec_sep,thous_sep)`
: to check if given string '`str`' is valid decimal number with precision '`m`' and scale '`n`', and with decimal separator '`dec_sep`' and thousand separator '`thous_sep`',

`is_valid_date(str,format)`
`is_valid_datetime(str,format)`
`is_valid_timestamp(str,format)`
: to check if the given string '`str`' is valid date and time data type in specified '`format`',

`is_valid_ipv4(str)`
`is_valid_ipv6(str)`
: *(since EVL 2.4)*
to check whether the string '`str`' is valid IPv4 or IPv6.

14.10 Checksum Functions

`md5sum(str)`
`sha224sum(str)`
`sha256sum(str)`
`sha384sum(str)`
`sha512sum(str)` *(since EVL 1.0)*

these standard checksum functions can be used in mapping this way for example:

```
*out->anonymized_username = sha256sum(*in->username);
```

When the argument is '`nullptr`', it returns '`nullptr`'. But in such case you need to use pointer manipulation, so the example would look like:

```
out->anonymized_username = sha256sum(in->username);
```

Functions headers:

```
std::string  md5sum(const char* const str);
std::string  md5sum(const std::string& str);
std::string* md5sum(const std::string* const str);

std::string  sha224sum(const char* const str);
std::string  sha224sum(const std::string& str);
std::string* sha224sum(const std::string* const str);

std::string  sha256sum(const char* const str);
std::string  sha256sum(const std::string& str);
std::string* sha256sum(const std::string* const str);

std::string  sha384sum(const char* const str);
std::string  sha384sum(const std::string& str);
std::string* sha384sum(const std::string* const str);

std::string  sha512sum(const char* const str);
std::string  sha512sum(const std::string& str);
std::string* sha512sum(const std::string* const str);
```

14.11 Mathematical Functions

`abs(x)` *(since EVL 2.8)*

`min(x)`
`max(x)` *(since EVL 2.8)*

`round(x)`
`ceil(x)`
`floor(x)`
`trunc(x)` *(since EVL 2.8)*

'`round`' to the nearest integer, when exactly in the middle (e.g. 2.5) round it up. '`ceil`' and '`floor`' round always down or up to the nearest integer. '`trunc`' truncates the fractional part.

```
*out->rounded   = round(2.56);    // 3.00
*out->ceiling   = ceil(2.56);     // 3.00
```

```
*out->floored   = floor(2.56);    // 2.00
*out->truncated = trunc(2.56);    // 2.00

*out->rounded   = round(-2.56);    // -3.00
*out->ceiling   = ceil(-2.56);     // -2.00
*out->floored   = floor(-2.56);    // -3.00
*out->truncated = trunc(-2.56);    // -2.00
```

When the argument is '`nullptr`', all functions return '`nullptr`'. But in such case you need to use pointer manipulation, so the example would look like:

```
out->rounded = round(in->value);
```

`pow(x)`
`sqrt(x)`

(since EVL 2.8)

14.12 Lookup Functions

For higher level overview check Section 15.1 [Lookup tables], page 199.

14.12.1 index, index_range, index_all, get_<type> *(since EVL 2.0)*

To avoid running lookup function several times to return different fields for given '`key_value(s)`' use '`index`' functions, which return only an index to the whole record found in a lookup.

Function '`index_all`' return all occurrences as a vector.

Functions '`get_<type>`' then return particular field value for given index.

`index(key_value(s))`
: to lookup by given '`key_value(s)`' and return an index of an occurrence. It doesn't care about the order of an occurrence, simply return that one which reach the first. Use better only when sure there is only one such value in a lookup, i.e. use with '`table::unique_key`' flag of '`table`' definition.

`index_range(key_value(s))`
: to lookup by given '`key_value(s)`', where the last key value is the one to fit within the range, and return an index of an occurrence. It doesn't care about the order of an occurrence, simply return that one which reach the first. Use better only when sure there is only one such value in a lookup, i.e. use with '`table::unique_key`' flag of '`table`' definition.

`index_all(key_value(s))`
: to lookup by given '`key_value(s)`' and return a vector of all occurrences.

`get_char(field_name,index)`
`get_uchar(field_name,index)`
`get_short(field_name,index)`
`get_ushort(field_name,index)`
`get_int(field_name,index)`
`get_uint(field_name,index)`
`get_long(field_name,index)`
`get_ulong(field_name,index)`
`get_int128(field_name,index)`
`get_uint128(field_name,index)`
`get_float(field_name,index)`
`get_double(field_name,index)`
`get_decimal(field_name,index)`
`get_date(field_name,index)`
`get_datetime(field_name,index)`
`get_timestamp(field_name,index)`
`get_time(field_name,index)`
`get_time_ns(field_name,index)`
`get_interval(field_name,index)`
`get_interval_ns(field_name,index)`
`get_string(field_name,index)`
`get_ustring(field_name,index)`

once having an '`index`' of the record, these functions return value of particular '`field_name`'.

Usage example

```
// get path to lookup dir from an environment
static string lookup_dir = std::getenv("LOOKUP_DIR");

// define a lookup table (file is sorted and binary, key is unique in the file)
static table CompanyGroupID(lookup_dir + "/DimCompany.CompanyGroupID.hist.evf",
        "generated/evd/Lookup/DimCompany.CompanyGroupID.hist.1.evd",
        "CompanyGroupID",
        table::unique_key);

// assign necessary fields
out->company_group_id = in->company_group_id;

// lookup and store as an vector
auto group_ids = CompanyGroupID.index_all(in->company_group_id);

// loop over such vector
for ( auto ind : group_ids ) {
  out->company_group = CompanyGroupID.get_ustring("CompanyGroupName",ind);
  out->company_id    = CompanyGroupID.get_int("CompanyID",ind);
  out->company_name  = CompanyGroupID.get_ustring("CompanyName",ind);
  add_record();  // produce a record for each Company in a group
}
discard();  // to avoid last record of the group to be doubled
```

14.12.2 lookup_<type> *(since EVL 1.0)*

`lookup_char(field_name,key_value(s))`
`lookup_uchar(field_name,key_value(s))`
`lookup_short(field_name,key_value(s))`
`lookup_ushort(field_name,key_value(s))`
`lookup_int(field_name,key_value(s))`
`lookup_uint(field_name,key_value(s))`
`lookup_long(field_name,key_value(s))`
`lookup_ulong(field_name,key_value(s))`
`lookup_int128(field_name,key_value(s))`
`lookup_uint128(field_name,key_value(s))`
`lookup_float(field_name,key_value(s))`
`lookup_double(field_name,key_value(s))`
`lookup_decimal(field_name,key_value(s))`
`lookup_date(field_name,key_value(s))`
`lookup_datetime(field_name,key_value(s))`
`lookup_timestamp(field_name,key_value(s))`
`lookup_time(field_name,key_value(s))`
`lookup_time_ns(field_name,key_value(s))`
`lookup_interval(field_name,key_value(s))`
`lookup_interval_ns(field_name,key_value(s))`
`lookup_string(field_name,key_value(s))`
`lookup_ustring(field_name,key_value(s))`

to lookup by given ‘`key_value(s)`’ and return value of ‘`field_name`’ of given data type.

Usage example

```
// define a lookup table (it is a sorted text file, ignore case of the key)
static table company("/data/dimensions/company.csv",
        "evd/dimensions/company.evd",
        "Company_ID",
        table::text_read | table::ignore_case);

// assign looked-up field
out->company_name = company.lookup_string("Name", in->company_group_id);
```

14.12.3 lookup_range_<type> *(since EVL 2.0)*

```
lookup_range_char(field_name,key_value(s))
lookup_range_uchar(field_name,key_value(s))
lookup_range_short(field_name,key_value(s))
lookup_range_ushort(field_name,key_value(s))
lookup_range_int(field_name,key_value(s))
lookup_range_uint(field_name,key_value(s))
lookup_range_long(field_name,key_value(s))
lookup_range_ulong(field_name,key_value(s))
lookup_range_int128(field_name,key_value(s))
lookup_range_uint128(field_name,key_value(s))
lookup_range_float(field_name,key_value(s))
lookup_range_double(field_name,key_value(s))
lookup_range_decimal(field_name,key_value(s))
lookup_range_date(field_name,key_value(s))
lookup_range_datetime(field_name,key_value(s))
lookup_range_timestamp(field_name,key_value(s))
lookup_range_time(field_name,key_value(s))
lookup_range_time_ns(field_name,key_value(s))
lookup_range_interval(field_name,key_value(s))
lookup_range_interval_ns(field_name,key_value(s))
lookup_range_string(field_name,key_value(s))
lookup_range_ustring(field_name,key_value(s))
```

to lookup by given '`key_value(s)`', where the last one is the one to fit within the range, and return '`field_name`' of given data type.

14.13 Other Functions

14.13.1 first_not_null *(since EVL 2.8)*

`first_not_null(value1,value2,...)`

return first object which is not null. Last value in the list can be fixed value.

For example following example can be used in mapping:

```
out->value = first_not_null(in->value_field1,
                  in->value_field2, in->value_field3);
```

which means that in output '`value`' will be assigned '`value_field1`' if it is not null, otherwise '`value_field2`' if it is not null, otherwise '`value_field3`' (even if it is null).

Example with default value:

```
out->value = first_not_null(in->value_field1,
                  in->value_field2, in->value_field3, "N/A");
```

which is the same as previous example, except in case also '`value_field3`' is null, then string '`N/A`' is assigned to '`*out->value`'.

14.13.2 getenv_<type> *(since EVL 2.8)*

To get environment variable into the mapping (i.e. in the '`evm`' file) standard C++ code can be used:

```
// This is not recommended example!
static const string release = std::getenv("RELEASE");
static int batch_id = atoi(std::getenv("BATCH_ID"));
```

```
*out->ID      = batch_id++;
*out->release = release;
```

However when there is no variable set in the environment, the failure of the mapping is not handled properly. So better use following EVL functions, which can use also default values.

> Always use '`static`' key word and in the case that the variable should not be changed in the mapping also use '`const`'.

`getenv_<integer_type>(<env_var>, [<default_value>])`

To read an environment variable as an integral type, there is also an optional argument with the value to be used in the case that the variable is not set. So for example

```
static int batch_id = getenv_int("BATCH_ID");
```

will fail in case of undefined '`BATCH_ID`' variable, but

```
static int batch_id = getenv_int("BATCH_ID",1);
```

will use number 1 in such case and do not fail.

`getenv_float(<env_var>, [<default_value>])`
`getenv_double(<env_var>, [<default_value>])`

For float types it behaves the same as for integral types, example with default value:

```
static float accumulate = getenv_float("START_VALUE",1000.00);
```

`getenv_decimal(<env_var>, <scale>, [<default_value>])`

For decimal there must be `<scale>` specified, e.g.:

```
static decimal X = getenv_decimal("X_VALUE",2);
```

except that it behaves the same as for integral types.

Example with default value:

```
static decimal X = getenv_decimal("X_VALUE",2,decimal(1000,2));
```

`getenv_<datetime_type>(<env_var>, [<format>], [<default_value>])`

There is a second optional argument `<format>` for date and time types which specifies in which format is the value of the environment variable. By default it uses format from environment variables `EVL_DEFAULT_<type>_PATTERN` which are by default set to

```
EVL_DEFAULT_DATE_PATTERN="%Y-%m-%d"
EVL_DEFAULT_DATETIME_PATTERN="%Y-%m-%d %H:%M:%S"
EVL_DEFAULT_TIMESTAMP_PATTERN="%Y-%m-%d %H:%M:%E*S"
EVL_DEFAULT_TIME_PATTERN="%H:%M:%S"
EVL_DEFAULT_TIME_NANO_PATTERN="%H:%M:%E*S"
```

So for example having environment variable `CURRENT_TIMESTAMP=20250319154034` would be used in mapping like this:

```
static const datetime curr_datetime =
    getenv_datetime("CURRENT_TIMESTAMP","%Y%m%d%H%M%S");
```

and with default value it would be:

```
static const datetime curr_datetime =
    getenv_datetime("CURRENT_TIMESTAMP",
                    "%Y%m%d%H%M%S",
                    datetime("2000-01-01 00:00:00"));
```

`getenv_string(<env_var>, [<default_value>])`
`getenv_ustring(<env_var>, [<default_value>])`

String types has second argument optional and can specify the default value in case the variable is not set in environment. Example with default value:

```
static const ustring author_name =
    getenv_ustring("AUTHOR",u"Jan Štěpnička");
```

The error message in case of undefined environment variable (and no default value) is

```
fail("Environment variable " + env_var + " is not set.")
```

Example of mapping using getenv_<type> function

```
static int batch_id = getenv_int("BATCH_ID");
static const uchar initial_load_flag = getenv_int("INITIAL_LOAD",0);
static const string release = getenv_string("RELEASE","no_release");

*out->ID      = batch_id++;
if (initial_load_flag)
  out->release = nullptr;
else
 *out->release = release;
```

The error message in case of undefined `BATCH_ID` would be

```
Environment variable BATCH_ID is not set.
```

15 Joins and Lookups

15.1 Lookup tables

15.1.1 Declaration and load

```
static table some_dimension("dim.csv", "dim.evd", "id", table::text_read);
```

where the third parameter is the comma separated list of key fields, i.e. the field(s) according to which it will look up.

Important: File has to be sorted according to key field!

Important: '`static`' is compulsory if you don't want to load lookup before processing each record.

The forth parameter is not mandatory and it may contain these flags:

`table::binary_read`
: the file is binary, default flag.

`table::text_read`
: the file is a text.

`table::unique_key`
: without this flag, it suppose there might be several keys and it will try to look for the first one. As it slows down looking up the data, if you know the lookup key is unique, use this flag.

`table::ignore_case`
: ignore case for strings.

More complex example:

```
static table some_lookup("input.csv", "input.evd", "key1, key2",
                  table::text_read | table::unique_key);
```

15.1.2 Methods

These methods looking up data in defined and loaded '`table`':

```
lookup_char
lookup_uchar
lookup_short
lookup_ushort
lookup_int
lookup_uint
lookup_long
lookup_ulong
lookup_float
lookup_double
lookup_decimal
lookup_date
lookup_timestamp
lookup_string
```

They are distinguished according to return value data type. They return the pointer to the value. For example '`lookup_char`' returns pointer to '`std::int8_t`' etc.

Methods return '`nullptr`' if there is no record found in the lookup table.

All methods have two or more parameters: first one is the name of the field to be returned and other parameters are values of the key fields defined by '`table`'.

Example: Let's have lookup (sorted) file `lookup_file.csv`:

```
1|2017-06-04|value1
1|2017-06-05|value2
2|2017-06-05|value3
```

with `lookup_file.evd`:

```
key1   int    sep="|"
key2   date   sep="|"
field3 string sep="\n"
```

Then in mapping (`evm` file):

```
static table lookup_file("lookup_file.csv", "lookup_file.evd", "key1, key2",
                   table::text_read | table::unique_key);

string* s = lookup_file.lookup_string("field3", 1, date("2017-06-05"));
// -> value2
```

16 Utils

EVL Utils are standalone command line utilities which can be split by the purpose.

CSV Utils

- Section 16.1 [csv2evd], page 201,
- Section 16.2 [csv2qvd], page 204,

EVD Utils

- Section 16.1 [csv2evd], page 201,
- Section 16.3 [evd2sql], page 206,
- Section 16.5 [json2evd], page 209,
- Section 16.8 [qvd2evd], page 215,
- Section 16.6 [pg2evd], page 211,

JSON Utils

- Section 16.5 [json2evd], page 209,

QVD Utils

- Section 16.2 [csv2qvd], page 204,
- Section 16.7 [qvd2csv], page 212,
- Section 16.8 [qvd2evd], page 215,
- Section 16.10 [qvd-header], page 217,

Other Utils

- Section 16.9 [evl increment run id], page 216, – increment and return unique job run ID, each job invocation will have it's unique ID
- Section 16.4 [guess timestamp format], page 208, – read an input dates, datetimes, timestamps and guess format string

16.1 csv2evd *(since EVL 2.2)*

Read `<file.csv>` or standard input, and guess:

- data types,
- field separator (unless option '`--separator=<char>`' is used),
- if strings are quoted (unless option '`--quote=<char>`' or '`--optional-quote=<char>`' is used),
- end-of-line character(s) (unless option '`--dos-eol`' or '`--lin-eol`' or '`--mac-eol`' is used)

and write EVD to standard output or to `<file.evd>`.

It uses header line for field names, spaces are replaced by underscores.

Separator is trying to be guessed in this order: '`,`' (comma), '`;`' (semi-colon), '`|`' (pipe), '`\t`' (tab), '`:`' (colon), '' (space).

Quotation character is guessed in this order: double quotes, single quotes.

EVD is EVL data definition file, for details see man 5 evd.

Synopsis

```
csv2evd
  [<file.csv>] [-o|--output=<file.evd>]
  [--inline]
  [-d|--date=<format>]
  [-h|--header=<field_name>,...]
  [-n|--no-header]
  [-l|--null=<string>]
  [-q|--quote=<char> | --optional-quote=<char>]
  [-s|--separator=<char>]
  [-t|--datetime=<format>]
  [--timestamp=<format>]
  [--dos-eol | --lin-eol | --mac-eol]
  [-v|--verbose]

csv2evd
  ( --help | --usage | --version )
```

Options

`-d, --date=<format>`
: by default it tries only '`%Y-%m-%d`', then '`%d.%m.%Y`'

`-h, --header=<field_name>,...`
: use comma separated list of field names instead of header line, for example when there is no header in csv file (option '`-n`' must be used) or when other field names would be used

`--inline`
: output EVD in the inline format (for example to use EVD by other component with '`-d`' option)

`--dos-eol`
: do not guess end-of-line character(s), but suppose the input is text with CRLF as end of line,

`--lin-eol`
: do not guess end-of-line character(s), but suppose the input is text with LF as end of line

`--mac-eol`
: do not guess end-of-line character(s), but suppose the input is text with CR as end of line

`-n, --no-header`
: with this option it suppose there is no header. Fields will be named '`field_001`', '`field_002`', etc.

`-l, --null=<string>`
: to specify what string is used for NULL values in CSV, empty string is allowed

`-o, --output=<file.evd>`
: write output into file <file.evd> instead of standard output

`--optional-quote=<char>`
: suppose optional quote character `<char>`, must be used together with '`--separator`'

`-q, --quote=<char>`
: do not guess if fields are quoted, but suppose `<char>` as quotation character

`-s, --separator=<char>`
: do not guess the separator, but use <char> instead

`-t, --datetime=<format>`
: by default it tries only '`%Y-%m-%d %H:%M:%S`'

`--timestamp=<format>`
: by default it tries only '`%Y-%m-%d %H:%M:%S.%E*f`'

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Having table.csv:

   ```
   id;started;value
   1;2019-06-06;some string
   ```

   This command:

   ```
   csv2evd table.csv
   ```

   will try to guess data types, field separator and if strings are quoted or not, and use header line for field names, to produce EVD to standard output:

   ```
   id       int                null="" sep=";"
   started  date("%Y-%m-%d")  null="" sep=";"
   value    string             null="" sep="\n"
   ```

2. Just an alternative invocation forwording output EVD to a file:

   ```
   csv2evd < table.csv > table.evd
   ```

3. To skip header and use different field names:

   ```
   csv2evd --header="first_field,other_field,last_one" \
     table.csv > table.evd
   ```

4. Case when there is no header in CSV file, but use specified field names:

   ```
   csv2evd --no-header --header="first_field,other_field,last_one" \
     table.csv > table.evd
   ```

5. No header in CSV and use generated field names '`field_001`', '`field_002`', etc.:

   ```
   csv2evd --no-header table.csv > table.evd
   ```

6. Consider specific date format, here day of year ('`001..366`'), and '`|`' as a field separator:

   ```
   csv2evd --date="%j" -s '|' table.csv > table.evd
   ```

16.2 csv2qvd *(since EVL 2.2)*

Read `<file.csv>` with sturcture defined either in `<evd>` file or by `<inline_evd>` or guess

- data types,
- field separator (unless option '`--separator=<char>`' is used),
- if strings are quoted (unless option '`--quote=<char>`' or '`--optional-quote=<char>`' is used),
- end-of-line character(s) (unless option '`--dos-eol`' or '`--lin-eol`' or '`--mac-eol`' is used)

and write QVD file to `<file.qvd>` or standard output. For guessing data types (EVD) it uses utility '`csv2evd`'.

EVD is EVL data definition file, for details see man 5 evd.

Synopsis

```
csv2qvd
  <file.csv>
  [-o|--output=<file.qvd>]
  [-d|--date=<format>]
  [-h|--header=<field_name>,...]
  [-n|--no-header]
  [-l|--null=<string>]
  [-q|--quote=<char> | --optional-quote=<char>]
  [-s|--separator=<char>]
  [-t|--datetime=<format>]
  [--timestamp=<format>]
  [--dos-eol | --lin-eol |--mac-eol]
  [-v|--verbose]

csv2qvd
  <file.csv> (<evd>|-d <inline_evd>)
  [-o|--output=<file.qvd>]
  [--dos-eol | --lin-eol |--mac-eol]
  [-v|--verbose]

csv2qvd
  ( --help | --usage | --version )
```

Options

Standard options:

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented to use already defined EVD

`--dos-eol`
: do not guess end-of-line character(s), but suppose the input is text with CRLF as end of line,

`--lin-eol`
: do not guess end-of-line character(s), but suppose the input is text with LF as end of line,

`--mac-eol`
: do not guess end-of-line character(s), but suppose the input is text with CR as end of line,

`-o, --output=<file.qvd>`
: write output into `<file.qvd>` instead of standard output

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

EVD options:

`--date=<format>`
: by default it tries only '`%Y-%m-%d`', then '`%d.%m.%Y`'

`-h, --header=<field_name>,...`
: use comma separated list of field names instead of header line, for example when there is no header in csv file (option '`-n`' must be used) or when other field names should be used

`-n, --no-header`
: with this option it suppose there is no header. Fields will be named '`field_001`', '`field_002`', etc.

`-l, --null=<string>`
: to specify what string is used for NULL values in CSV, empty string is allowed

`--optional-quote=<char>`
: suppose optional quote character `<char>`, must be used together with '`--separator`'

`-q, --quote=<char>`
: do not guess if fields are quoted, but suppose `<char>` as quotation character

`-s, --separator=<char>`
: do not guess the separator, but use `<char>` instead

`-t, --datetime=<format>`
: by default it tries only '`%Y-%m-%d %H:%M:%S`'

`--timestamp=<format>`
: by default it tries only '`%Y-%m-%d %H:%M:%S.%E*f`'

Examples

1. Having '`some.csv`':

   ```
   id;started;value
   1;2019-06-06;some string
   ```

   The command:

   ```
   csv2qvd --null="NULL" some.csv > some.qvd
   ```

   will produce some.qvd file with these field:

   ```
   id       int               null="NULL"  sep=";"
   ```

```
started  date("%Y-%m-%d")  null="NULL"  sep=";"
value    string            null="NULL"  sep="\n"
```

2. Following invocation will guess data types, field separator and if strings are quoted or not, and use header line for field names:

   ```
   csv2qvd table.csv > table.qvd
   ```

   With the '`--verbose`' option it will write to standard error the whole EVD file which was used:

   ```
   csv2qvd --verbose table.csv > table.qvd
   ```

3. To skip header and use different field names:

   ```
   csv2qvd --header="first_field,other_field,last_one"
     table.csv > table.qvd
   ```

4. Case when there is no header in CSV file, but use specified field names:

   ```
   csv2qvd --no-header --header="first_field,other_field,last_one" \
     table.csv > table.qvd
   ```

5. No header in CSV and use generated field names '`field_001`', '`field_002`', etc.:

   ```
   csv2qvd --no-header table.csv > table.qvd
   ```

6. Consider specific date format, here day of year ('`001..366`'), and '`|`' as a field separator:

   ```
   csv2qvd --date="%j" -s '|' table.csv > table.qvd
   ```

7. To use own (specific or already generated) EVD file (i.e. data types definition):

   ```
   csv2qvd table.csv table.evd > table.qvd
   ```

16.3 evd2sql *(since EVL 2.6)*

Read the EVL data definition (a.k.a. EVD) from `<table.evd>` and write to standard output (unless '`--output`' option is used) '`CREATE TABLE`' statement specific for given SQL dialect: ANSI, MS SQL, PostgreSQL, Redshift, etc.

When more than one `<table.evd>` files specified, then write to '`*.sql`' files named accordingly with the same basename.

The SQL statement looks like this in general:

```
CREATE TABLE [IF NOT EXISTS] ["<schema_name>".]"<table_name>" (
    <column_1_based_on_evd>
  , <column_2_based_on_evd>
  , <column_3_based_on_evd>
  , ...
[<table_constraints>]
)
[<table_attributes>]
;
```

EVL data types mapping:

EVL	**Postgres/Redshift**	**MS SQL**
'`char`'	'`"CHAR"`'	'`SMALLINT`'
'`uchar`'	'`BOOLEAN`'	'`TINYINT`'
'`short`'	'`SMALLINT`'	'`SMALLINT`'
'`ushort`'	'`SMALLSERIAL`'	'`INT`'
'`int`'	'`INTEGER`'	'`INT`'
'`uint`'	'`SERIAL`'	'`BIGINT`'
'`long`'	'`BIGINT`'	'`BIGINT`'

'ulong'	'BIGSERIAL'	'DECIMAL(20,0)'
'int128'	'NUMERIC(38,0)'	'DECIMAL(38,0)'
'utint128'	'NUMERIC(38,0)'	'DECIMAL(38,0)'
'float'	'REAL'	'REAL'
'double'	'DOUBLE PRECISION'	'FLOAT'
'decimal(m,n)'	'NUMERIC(m,n)'	'DECIMAL(m,n)'
'string'	'TEXT'	'VARCHAR'
'ustring'	'TEXT'	'NVARCHAR'
'date'	'DATE'	'DATE'
'time'	'TIME'	'TIME'
'interval'	'INTERVAL'	N/A
'datetime'	'TIMESTAMP(0)'	'DATETIME2(0)'
'timestamp'	'TIMESTAMP(6)'	'DATETIME2(6)'

Synopsis

```
evd2sql
  ( <table.evd>... | -i|--input <table.evd> )
  [-d|--sql-dialect <database> ]
  [--if-not-exists]
  [-o|--output ( <table.sql> | <target_dir> ) ]
  [-s|--schema <schema_name>]
  [-t|--table <table_name>]
  [--table-attributes <table_attributes>]
  [--table-constraints <table_constraints>]
  [--varchar <length>]
  [-v|--verbose]

evd2sql
  ( --help | --usage | --version )
```

Options

`-d, --sql-dialect=<database>`
: currently these SQL types are supported:

```
ansi (default)
mssql
postgres
redshift
```

`--if-not-exists`
: use 'CREATE TABLE IF NOT EXISTS' instead of default 'CREATE TABLE'

`-i, --input=<table.evd>`
: read file `<table.evd>`

`-o, --output=<path>`
: if `<path>` is an existing directory, it writes output there. If it is not a directory, it is considered as an output file name.

`-s, --schema=<schema_name>`
: add `<schema_name>` to table name

`-t, --table=<table_name>`
: by default basename of `<table.evd>` from '`--input`' option is used as table name in '`CREATE TABLE`' statement, this option can overwrite it. When reading EVD from standard input, this option is recommended, otherwise table name will be empty

`--table-attributes=<table_attributes>`
: string to be added right after closing bracket, e.g. for Redshift it might be '`SORTKEY (some_id,other_col)`'

`--table-constraints=<table_constraints>`
: string to be added right after column list, e.g. '`, PRIMARY KEY (some_id)`'

`--varchar=<length>`
: specify the default VARCHAR length, default is 256

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Having an EVD file '`some.evd`':

   ```
   id       int             sep=";"
   started  date    null="" sep=";"
   value    string  null="" sep="\n"
   ```

   This command:

   ```
   evd2sql -s postgres -i some.evd --if-not-exists
   ```

   will produce:

   ```
   CREATE TABLE IF NOT EXISTS "some" (
       id      INTEGER NOT NULL
     , started DATE
     , value   TEXT
   );
   ```

16.4 guess-timestamp-format *(since EVL 2.4)*

Read line by line standard input or an input `<file>` and try to guess format string of the date, datetime or timestamp.

It uses the `<config_file>` with the list of format strings like:

```
%Y-%m-%d %H:%M:%S
%Y-%m-%dT%H:%M:%S
%Y-%m-%d
%d/%m/%y %H:%M
%-m/%-d/%y %H:%M
%d.%m.%y %H:%M
%d.%m.%y %-H:%M
```

```
%d.%m.%y %-H:%M:%S
```

Unless '`--config`' option is used, it uses a file timestamp-formats-order.csv from the same folder as this script (try '`which guess-timestamp-format`').

Synopsis

```
guess-timestamp-format
  [-i|--input=<file>] [-c|--config=<config_file>] [-d|--with-data-type]
  [-v|--verbose]

guess-timestamp-format
  ( --help | --usage | --version )
```

Options

-c, –config=<config_file>

-d, –with-data-type

-i, –input=<file>

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Let us have this file '`timestamps.csv`':

   ```
   03/12/2022 11:20:00
   03/12/2022 01:02:00
   03/13/2022 03:24:55
   03/14/2022 11:20:59
   ```

   Following comman recognize the format:

   ```
   guess-timestamp-format < timestamps.csv
   ```

   and returns:

   ```
   %m/%d/%Y %H:%M:%S
   ```

   With the option '`--with-data-type`' it returns full data type information:

   ```
   datetime("%m/%d/%Y %H:%M:%S")
   ```

16.5 json2evd *(since EVL 2.4)*

Read `<file.json>` or standard input, guess data types, and write EVD to standard output or to `<file.evd>`.

Synopsis

```
json2evd
  [<file.json>] [-o|--output=<file.evd>]
  [-d|--date=<format>]
  [-l|--null=<string>]
  [-q|--quote=<char>]
  [-s|--separator=<char>]
  [-t|--datetime=<format>]
  [-v|--verbose]

json2evd
  ( --help | --usage | --version )
```

Options

`-d, --date=<format>`
: by default it tries only '`%Y-%m-%d`', then '`%Y%m%d`', then '`%d.%m.%Y`'

`-l, --null=<string>`
: to specify what string is used for NULL values in JSON, empty string is allowed

`-o, --output=<file.evd>`
: write output into file <file.evd> instead of standard output

`-q, --quote=<char>`
: do not guess if fields are quoted, but suppose <char> as quotation character

`-s, --separator=<char>`
: do not guess the separator, but use <char> instead

`-t, --datetime=<format>`
: by default it tries only '`%Y-%m-%d %H:%M:%S`', then '`%Y%m%d%H%M%S`'

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Having some.json:

   ```
   TBA
   ```

   This command:

   ```
   json2evd --null="" some.json
   ```

   will produce:

   ```
   id       int             null="" sep=";"
   started  date("%Y-%m-%d") null="" sep=";"
   value    string          null="" sep="\n"
   ```

16.6 pg2evd *(since EVL 2.6)*

Read the definition of PostgreSQL table and write EVD to standard output or to `<file>`.

Password is taken:

1. from file '`$EVL_PASSFILE`', which is by default '`$HOME/.evlpass`',
2. from file '`$PGPASSFILE`', which is by default '`$HOME/.pgpass`'.

When such file has not permissions 600, it is ignored! For details see '`evl-password`'.

Synopsis

```
pg2evd
  [<schema>.]<table> [-o|--output=<file>] [-f|--output-format (evd|json)]
  [-b|--dbname=<database>] [-h|--host=<hostname>] [-p|--port=<port>]
  [-u|--username=<pguser>] [--psql=<psql_options>]
  [-d|--date=<format>]
  [-l|--null[=<string>]]
  [-q|--quote=<char>]
  [-r|--record-separator=<char>]
  [-s|--field-separator=<char>]
  [-t|--datetime=<format>]
  [-v|--verbose]

pg2evd
  ( --help | --usage | --version )
```

Options

`-d, --date=<format>`
: by default it produce no format for date, so the EVL_DEFAULT_DATE_PATTERN is then used (by default it is '`"%Y-%m-%d"`')

`-l, --null=<string>`
: add '`null="<string>"`' to every field

`-o, --output=<file>`
: write output into file <file> instead of standard output

`-f, --output-format=(evd|json)`
: write output in given file format, by default write '`evd`'

`-q, --quote=<char>`
: add '`quote="<char>"`' to every field

-s, –separator=<char>

`-r, --record-separator=<char>`
: use '`sep="<char>"`' for last field

`-s, --field-separator=<char>`
: add '`sep="<char>"`' to each field, except the last one

`-t, --datetime=<format>`
: by default it produces no format for datetime, so the EVL_DEFAULT_DATETIME_PATTERN is then used (which is by default set to '`"%Y-%m-%d %H:%M:%S"`')

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

'psql' options:

`-b, --dbname=<database>`
: either this or environment variable '`PGDATABASE`' should be provided, if not, then current system username is used as psql database. If also '`PGDATABASE`' environment variable is set, this option has preference. (This option is provided to '`psql`' command.)

`-h, --host=<hostname>`
: either this or environment variable '`PGHOST`' should be provided when connecting to other host than localhost. If also '`PGHOST`' variable is set, this option has preference. (This option is provided to '`psql`' command.)

`-p, --port=<port>`
: either this or environment variable '`PGPORT`' should be provided when using other then standard port '`5432`'. (This option is provided to '`psql`' command.)

`--psql=<psql_options>`
: all other options to be provides to psql command. See '`man psql`' for details.

`-u, --username=<pguser>`
: either this or environment variable '`PGUSER`' should be provided, if not, then current system username is used as psql user. If variable '`PGUSER`' is set, this option has preference. (This option is provided to '`psql`' command.)

Examples

1. Having '`some_table`' in databese '`some_db`':

   ```
   id      integer
   started date
   value   varchar(20)
   ```

   This command:

   ```
   pg2evd --null="" --separator=";" some_db.some_table
   ```

   will produce:

   ```
   id       int     null="" sep=";"
   started  date    null="" sep=";"
   value    string  null="" sep="\n"
   ```

16.7 qvd2csv *(since EVL 2.4)*

Read `<file.qvd>` and write CSV file to `<file.csv>` or standard output. It uses data types from QVD header or from existing `<evd>` file or from `<inline_evd>`.

EVD is EVL data definition file, for details see man 5 evd.

Synopsis

```
qvd2csv
  <file.qvd>
  [-o|--output=<file.csv>]
  [--all-as-string | --real-as-decimal[=<precision>,<scale>]]
  [-d|--date=<format>]
  [-h|--header=<field_name>,...]
  [-n|--no-header]
  [-l|--null=<string>]
  [-q|--quote=<char>]
  [-s|--separator=<char>]
  [-t|--datetime=<format>]
  [-a|--dos-eol | -b|--mac-eol]
  [--filter=<condition>]
  [--first-record=<n>]
  [--guess-uniform-symbol-size]
  [--low-memory]
  [-v|--verbose]

qvd2csv
  <file.qvd> (<evd>|-d <inline_evd>)
  [-m|--match-fields]
  [-o|--output=<file.csv>]
  [-h|--header=<field_name>,...]
  [-n|--no-header]
  [-a|--dos-eol | -b|--mac-eol]
  [--filter=<condition>]
  [--first-record=<n>]
  [--guess-uniform-symbol-size]
  [--low-memory]
  [-v|--verbose]

qvd2csv
  ( --help | --usage | --version )
```

Options

`--all-as-string`
: interpret all fields as strings. (Since EVL 2.5.)

`-d, --data-definition=<inline_evd>`
: either this option or the file `<evd>` must be presented to use already defined (custom) EVD

`-a, --dos-eol`
: output DOS end-of-line, i.e. CR+LF (‘`\r\n`’)

`-b, --mac-eol`
: output Mac end-of-line, i.e. CR (‘`\r`’)

`--date=<format>`
: to specify a `<format>` for date data type

`--filter=<condition>`
: read only records with given `<condition>`. (Since EVL 2.6.)

`--first-record=<n>`
: start to read from the record number `<n>`. (Since EVL 2.6.)

`--guess-uniform-symbol-size`
: might speed up indexing of dictionary, but it could not work in all cases. Use only in special cases when need really good performance. (Since EVL 2.6.)

`-h, --header=<field_name>,...`
: use comma separated list of field names instead of header line, for example when you don't want to use field names from QVD header.

`--low-memory`
: do not read dictionary into memory. This could save memory consumption, but slows down reading the source file. (Since EVL 2.6.)

`-l, --null=<string>`
: to specify what string is used for NULL values in CSV, empty string is allowed

`-m, --match-fields`
: to read only a subset of fields from QVD file or to read them in different order

`-n, --no-header`
: with this option it produces no header line

`-o, --output=<file.csv>`
: write output into `<file.csv>` instead of standard output

`-q, --quote=<char>`
: to use quoted fields for the CSV output. When data contains such `<char>`, all of them are escaped by duplicating them. For example using '`--quote="\""`' will serve data like '`some "text"`' as '`"some ""text"""`'.

`--real-as-decimal[=<precision>,<scale>]`
: interpret '`real`' data types as '`decimal(<precision>,<scale>)`'. When no `<precision>` or `<scale>` is specified, use values from environment variables '`EVL_DEFAULT_DECIMAL_PRECISION`' and '`EVL_DEFAULT_DECIMAL_SCALE`', which are by default set to 18 and 2. (Since EVL 2.5.)

`-s, --separator=<char>`
: to use `<char>` as field separator for the CSV output

`-t, --datetime=<format>`
: to specify a `<format>` for datetime data type

`-v, --verbose`
: print to standard error output info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Having '`some.qvd`', the command to produce CSV file with empty strings representing NULL values, dates in format '`DD.MM.YYYY`' and with Windows end-of-line (i.e. CRLF):

   ```
   qvd2csv --null="" --date="%d.%m.%Y" --dos-eol some.qvd > some.csv
   ```

2. To filter only particular records from ‘`large.qvd`’, for example we would like to read only latest records represented by field ‘`invoice_id`’:

   ```
   qvd2csv --filter="invoice_id>7654000" large.qvd > latest.csv
   ```

3. To cut only particular columns from ‘`large.qvd`’, for example only column ‘`invoice_id`’:

   ```
   qvd2csv --match-fields -d 'invoice_id int null=""' large.qvd > latest.csv
   ```

4. To read only after by some number of rows:

   ```
   qvd2csv --first-record=1234000 huge.qvd > latest.csv
   ```

   This could be quite useful when reading a huge QVD file.

16.8 qvd2evd *(since EVL 2.4)*

Read header of `<file.qvd>` or standard input, guess data types, and write EVD to standard output or to `<file.evd>`.

EVD is EVL data definition file, for details see man 5 evd.

Synopsis

```
qvd2evd
  [<file.qvd>] [-o|--output=<file.evd>]
  [--all-as-string | --real-as-decimal[=<precision>,<scale>]]
  [-d|--date=<format>]
  [--inline]
  [-l|--null[=<string>]]
  [-q|--quote=<char>]
  [-r|--record-separator=<char>]
  [-s|--field-separator=<char>]
  [-t|--datetime=<format>]
  [-v|--verbose]

qvd2evd
  ( --help | --usage | --version )
```

Options

`--all-as-string`
: produce EVD with all fields as strings. (Since EVL 2.5.)

`-d, --date=<date_format>`
: use format argument for date data type

`--inline`
: output EVD in the inline format (for example to use EVD by other component with ‘`-d`’ option)

`-l, --null=<string>`
: to specify what string is used for NULL values in QVD, empty string is allowed

`-o, --output=<file.evd>`
: write output into file <file.evd> instead of standard output

`-q, --quote=<char>`
: to use a quote argument in EVD

`--real-as-decimal[=<precision>,<scale>]`
: produce EVD with '`decimal(<precision>,<scale>)`' instead of '`double`'. When no `<precision>` or `<scale>` is specified, it uses values from environment variables '`EVL_DEFAULT_DECIMAL_PRECISION`' and '`EVL_DEFAULT_DECIMAL_SCALE`', which are by default set to 18 and 2. (Since EVL 2.5.)

`-r, --record-separator=<char>`
: use '`sep="<char>"`' for last field

`-s, --field-separator=<char>`
: add '`sep="<char>"`' to each field, except the last one

`-t, --datetime=<format>`
: use format for datetime data type by default it produces no format for datetime, so the EVL_DEFAULT_DATETIME_PATTERN is then used (which is by default set to '`"%Y-%m-%d %H:%M:%S"`')

`-v, --verbose`
: print to STDERR info/debug messages

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`--version`
: print version and exit

Examples

1. Having '`some.qvd`', this command:

   ```
   qvd2evd --null -r '\n' -s ';' -d '%d.%m.%Y' some.qvd
   ```

   will produce:

   ```
   id       int                null="" sep=";"
   started  date("%d.%m.%Y") null="" sep=";"
   value    string             null="" sep="\n"
   ```

16.9 evl_increment_run_id *(since EVL 2.0)*

Let's have defined variable `EVL_RUN_ID_FILE` with the path to the file which contains the high watermark of some incremental ID. (Let's call such value `RUN_ID` as the purpose is usually having unique ID of each job.)

The invocation of the command '`evl_increment_run_id`' takes the value from the file defined by `EVL_RUN_ID_FILE`, increase it by one and write it back to file and also to '`stdout`'.

So one can use it this way:

In each job, there would be

```
export EVL_RUN_ID=$(evl_increment_run_id)
```

and then the value can be used in the mapping for example:

```
static long run_id = atol(std::getenv("EVL_RUN_ID"));
```

Data type of `RUN_ID` is '`long`'.

If `EVL_RUN_ID_FILE` is not defined, then '`evl_increment_run_id`' command fail. If it is defined, but the file doesn't exist or is empty, then the file is created with the initial value~0.

When the file `EVL_RUN_ID_FILE` is locked by other process, ‘`evl_increment_run_id`’ will try every 300\,ms to get access. It will wait at most `EVL_RUN_ID_FILE_LOCK_WAIT` seconds before fail. Default is 30 seconds.

16.10 qvd-header *(since EVL 2.3)*

Take the header of `<file.qvd>` or standard input and produce to standard output particular information, for example EVL data definition file or number of records.

Synopsis

```
qvd-header
  [<file.qvd>] --output=evd
  [--all-as-string | --real-as-decimal[=<precision>,<scale>]]
  [-d|--date=<format>]
  [--inline]
  [-l|--null=<string>]
  [-q|--quote=<char>]
  [-r|--record-separator=<char>]
  [-s|--field-separator=<char>]
  [-t|--datetime=<format>]
  [-v|--verbose]

qvd-header
  [<file.qvd>] --output=(json|xml)
  [--fields]
  [-v|--verbose]

qvd-header
  [<file.qvd>]
  [ --table-name | --no-of-records | --fields | --tag=<xml_tag_name> ]
  [-v|--verbose]

qvd-header
  ( --help | --usage | --version )
```

Options

`--no-of-records`
: return the value of ‘`NoOfRecords`’ tag

`--fields`
: provide only fields’ information

`--table-name`
: return the value of ‘`TableName`’

`--tag=<xml_tag_name>`
: return the value of `<xml_tag_name>`

Output Options:

`--output=evd`
: return EVD data types definition

`--output=json`
: return information as JSON

`--output=xml`
: return information as XML

EVD options:

`--all-as-string`
: produce EVD with all fields as strings. (Since EVL 2.5.)

`-d, --date=<date_format>`
: use format argument for date data type

`--inline`
: output EVD, XML or JSON in the inline format (for example to use EVD by other component with '`-d`' option)

`-l, --null=<string>`
: to specify what string is used for NULL values in EVD

`-q, --quote=<char>`
: to use a quote argument in EVD

`--real-as-decimal[=<precision>,<scale>]`
: produce EVD with '`decimal(<precision>,<scale>)`' instead of '`double`'. When no `<precision>` or `<scale>` is specified, it uses values from environment variables '`EVL_DEFAULT_DECIMAL_PRECISION`' and '`EVL_DEFAULT_DECIMAL_SCALE`', which are by default set to 18 and 2. (Since EVL 2.5.)

`-r, --record-separator=<char>`
: use '`sep="<char>"`' for last field

`-s, --field-separator=<char>`
: use '`sep="<char>"`' for each field

`-t, --datetime=<date_format>`
: use format for datetime data type

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Examples

1. EVD example:

   ```
   qvd-header some.qvd --output=evd --record-separator="\n" \
                       --null="" --date="%Y-%m-%d"
   ```

   will produce for example:

   ```
   id          int                 null=""
   ```

```
some_stamp  datetime         null=""
some_date   date("%Y-%m-%d") null=""
value       string           null="" sep="\n"
```

2. JSON example:

   ```
   qvd-header some.qvd --output=json --fields
   ```

   will produce for example:

   ```
   {
     "fields":
     [
       {
         name: "REQUEST_HOUR",
         type: "timestamp",
         format: "%Y-%m-%d"
       },
       ...
     ]
   }
   ```

   And:

   ```
   qvd-header some.qvd --output=json
   ```

   will produce for example:

   ```
   {
     "name": "Table1",
     "records": 3615,
     "fields":
     [
       {
         name: "REQUEST_HOUR",
         type: "timestamp",
         format: "%Y-%m-%d",
         "tags": [
           "$numeric",
           "$timestamp"
         ]
       },
       ...
     ]
   }
   ```

EVM Functions Index

EVD Data Types Index

Variables Index

General Index