ETL Engine Documentation

Rook IT ApS

Rook IT ApS

Table of contents

1. ETL Engine

A high-performance ETL toolkit for transforming arbitrary health-data sources into the OMOP Common Data Model.

The ETL Engine consumes the output of existing OHDSI tooling — **Rabbit-in-a-Hat** and **Usagi** — and turns table-to-table mappings into a fully orchestrated transformation, so data owners can focus on the data rather than the plumbing.

Get started → | **Download** | **Mapping language reference →**

1.1 What it does

- Reads from CSV, **MariaDB** / **MySQL**, **PostgreSQL** (native or ODBC), or **SQL Server** sources and writes to any of the same as a CDM target — every driver works on both sides and combinations can be freely mixed.
- Runs your mappings straight from the **Rabbit-in-a-Hat** scan-report file: you write the transformation logic as annotations in the report itself — trivial where the source data is clean, as involved as it needs to be where it isn't — instead of building and maintaining a separate ETL application.
- Resolves vocabulary mappings via **Usagi** files and the **OMOP standard vocabulary** out of the box.
- Adds an expressive **mapping language** for the cases that go beyond simple field-to-field copies — joins, conditional chains, vocabulary lookups, multi-sized results, virtual fields.
- Ships as a **Docker** / **Podman image** for reproducible deployment.

1.2 Where to go next

If you are…	Start here
Installing the engine for the first time	Getting Started → Installation
Writing your first mapping	DSL Guide → Mapping Language Syntax
Learning by example	Examples — Synthmas walkthrough + pattern recipes
Looking up a specific rule	Rule Reference
Debugging a failed run	Troubleshooting
Configuring `etl.conf`	Configuration Reference

1.3 Licensing & support

The Free tier translates a Synthmas-sized dataset end-to-end — including table rules, joins, normalisation, the lot — on a single worker thread and a fixed record budget, with no licence file. RDBMS drivers, automation, multi-thread scaling, and unbounded record counts come with the Paid tier, which ships with a support package.

Read more →

1.4 About

The ETL Engine is built and maintained by Rook IT ApS, specialising in secure data transformation and high-performance computing for sensitive health data.

2. Licensing

The ETL Engine ships in two licensing modes: **Free** and **Paid**.

2.1 Tiers

2.1.1 Free

Run the engine for free, no license file required, on a fixed per-CDM-table row cap — enough to run a small dataset like Synthmas end-to-end, including table rules, joins, normalisation, and the full mapping language.

Capability	Free
CSV source → CSV target	✅
RDBMS drivers (PostgreSQL / MariaDB / MySQL / SQL Server)	❌
Field-level rules	✅
Table rules (`Join` , `LeftJoin` , `Normalize` , `LoopOver` , `DependOn` , `InsertRecord` , `TraceWhen`)	✅
Cross-run state persistence (`StoreIndexMap` , `StoreSequence`)	❌
Automation (`create omop cdm` , `populate vocabulary` , era tables)	❌
Worker threads	1
Records per CDM table	up to 150,000

The Free tier is enough to run the bundled **Synthmas** dataset end-to-end — a synthetic Massachusetts patient population generated by Synthea, with a Rabbit-in-a-Hat file that exercises real CDM tables, joins, vocabulary lookups, and date-arithmetic patterns. Three of the larger tables (`observation` , `measurement` , `procedure_occurrence`) hit the cap and have their tail rows dropped; the other dozen CDM tables populate in full. Grab the bundle from rook-it.com/downloads and follow the Synthmas walkthrough.

What the cap looks like in practice — why the `run.log` counter can climb past 150,000, and how Paid-only features are reported when you hit them — is covered in the Synthmas walkthrough and its FAQ.

The Paid tier removes the cap and unlocks every feature listed above.

2.1.2 Paid

A signed license file unlocks the full engine.

Capability	Paid
Everything the Free tier offers	✅
RDBMS drivers (PostgreSQL, MariaDB / MySQL, SQL Server)	✅
Automation (CDM setup, vocabulary loading, era tables, preceding-visit IDs)	✅
Worker threads	unlimited (license-configurable)
Records translated per run	unlimited
Email support	included

The license file lives alongside `etl.conf` (typically at `etc/etl-engine/license.lic`). The shipped `bin/run-etl-engine.sh` mounts that directory into the container automatically.

A license may carry an expiry date. The expiry date is **inclusive** — the license is valid through the whole of that calendar day (local time) and lapses at the start of the following day, so a license that expires `2030-12-31` still runs on 31 December 2030. When a license has lapsed the engine keeps running but drops to the Free tier (the banner prints `UNLICENSED`), rather than refusing to start.

2.2 Contact

For Paid licensing, support, or anything else (the Synthmas bundle itself is a free download from rook-it.com/downloads, no contact needed):

Get in touch via rook-it.com

— or email **info@rook-it.com** directly.

2.3 See also

- Third-party licenses — separate from the ETL Engine's own commercial licensing, this page lists the open-source libraries the engine bundles and their respective licenses (legally required attribution).

3. Guide

3.1 Getting Started

3.1.1 Getting Started

This section walks through installing the ETL Engine, understanding the folder layout it expects, and running a first transformation.

The ETL Engine has five end-point drivers. **Every driver works on either side** — source, CDM, or both — and source/CDM combinations can be freely mixed.

- **CSV**
- **MariaDB / MySQL**
- **PostgreSQL** (native libpq)
- **PostgreSQL over ODBC** (`psql-odbc`)
- **SQL Server** (over ODBC)

Pages

- Installation — installing the Docker image.
- Folder Layout — the directories the engine expects.
- First ETL Run — running the engine end-to-end.

3.1.2 Installation

The ETL Engine is shipped as a `tar.gz` package that contains:

- a Docker image (`data/etl-engine.docker`),
- install and run scripts (`bin/install-etl-engine.sh`, `bin/run-etl-engine.sh`),
- a configuration template (`etc/etl-engine/etl.conf`),
- the empty data directories the engine expects (`etc/etl-engine.d/`),
- and a `logs/` directory for log output.

A typical artefact name is `etl-engine-@ETL_VERSION@.tar.gz` — the version number tracks engine releases.

The latest release is a free download from rook-it.com/downloads, no login. Older versions and debug builds are available on the same site behind a login for paid customers (older versions matter in healthcare deployments) and Rook IT consultants (debug builds).

Docker or Podman

The instructions throughout this documentation use `docker`, but `podman` works as a drop-in replacement — the install and run scripts auto-detect which one is available. When **both** are installed they prefer `docker`. To force a choice on a host that has both, set `ETL_CONTAINER_TOOL`:

```
ETL_CONTAINER_TOOL=podman ./bin/install-etl-engine.sh
```

Prerequisites

- A working **Docker** or **Podman** installation.
- A user account that can talk to the Docker daemon. The install script verifies this for you and fails with a clear message if not. Either run as `root` or add your user to the group that owns `/var/run/docker.sock`.

Install

Unzip the artefact into the directory of your choice, then run the install script:

```
$ tar xzf etl-engine-@ETL_VERSION@.tar.gz
$ cd etl-engine-@ETL_VERSION@
$ ./bin/install-etl-engine.sh
```

The script does one thing: it loads the bundled image (`docker load -i data/etl-engine.docker`) into your local Docker registry under the tag `etl-engine`. There are no other side effects — no system packages installed, no system services configured.

When the script returns, the engine is ready to be configured. The prepared folder structure also appears around the install location — see Folder Layout.

Run

Once the configuration file and a Rabbit-in-a-Hat file are in place, run the engine:

```
$ cd etl-engine-@ETL_VERSION@
$ ./bin/run-etl-engine.sh
```

The run script clears any previous logs, starts the `etl-engine` container with the standard volume mounts, and lets the engine run to completion in the foreground.

The container is started with elevated privileges so that performance profiling tools (`/sys/kernel/debug` access, host PID namespace) can attach if needed. If your platform's security policy disallows that, adjust the run script — see First ETL Run for what each flag does.

Upgrade

To upgrade, replace the artefact and re-run the install script:

```
$ tar xzf etl-engine-@ETL_VERSION@.tar.gz
$ cd etl-engine-@ETL_VERSION@/bin
$ ./install-etl-engine.sh
```

The new image overwrites the previous `etl-engine` tag. Your configuration and data directories are not touched.

3.1.3 First ETL Run

This page walks through the prerequisites and first execution of the ETL Engine. It assumes the engine has already been installed and the folder layout is intact.

Quick path — the Synthmas demo

The Synthmas demo runs on top of a base engine install, layered in via a small **overlay tarball** that adds the Rabbit-in-a-Hat file, the Usagi files it references, a Synthmas-tuned `etl.conf`, and a script that pulls the source CSVs from MITRE.

Both tarballs — the latest engine and the Synthmas overlay — are free downloads at rook-it.com/downloads (no login). Older engine versions and debug builds live behind a login on the same site for paid customers and Rook IT consultants.

From a fresh download of both tarballs to a populated CDM:

```
$ tar xzf etl-engine-X.Y.Z.tar.gz                        # the engine
$ cd etl-engine-X.Y.Z
$ tar xzf ../synthmas-single-RiaH-X.Y.Z.tar.gz --strip-components=1   # overlay the demo
$ ./bin/install-etl-engine.sh                            # load the Docker image
$ ./bin/setup-synthmas.sh                                # pull source CSVs from MITRE
$ ./bin/check-vocabs.sh                                  # see which vocabularies the RiaH needs
$ ./bin/setup-vocab.sh ~/Downloads/vocabulary_*_v5.zip   # (optional) unpack an Athena vocab
$ ./bin/run-etl-engine.sh                                # run the engine
```

Output appears in `etc/etl-engine.d/cdm-out/` as a directory of CDM CSV files. The vocab step is optional — without it, `VocabMap` / `VocabSourceId` / `UsagiMap` rules leave their concept fields NULL but the rest of the CDM still populates.

See Vocabulary from Athena for the full walkthrough of selecting the right vocabularies (the Synthmas RiaH needs SNOMED, RxNorm, LOINC, and CVX).

The rest of this page covers the same flow in more detail, plus how to point the engine at your own data and Rabbit-in-a-Hat file.

Prerequisites

Before running the engine, make sure the following are in place:

1. **`etl.conf`** — under `etc/etl-engine/`. The shipped file is a working Free-tier template (CSV in / CSV out) — see the Configuration Reference for every key, and uncomment the Paid-tier blocks at the bottom of the file when you upgrade.
2. **A Rabbit-in-a-Hat file** — alongside `etl.conf`, named in the `rabbit configuration` block. The bundled Synthmas RiaH is the default; replace it when you have your own. See Rabbit-in-a-Hat Integration.
3. **(Optional) Usagi files** — in `etc/etl-engine.d/usagi/`, if any rule chain uses `UsagiMap`.
4. **OMOP vocabulary** — only needed if your rule chains do `VocabMap` / `VocabSourceId` / `UsagiMap` lookups. How to load it depends on the CDM driver:

- **CSV target (Free tier)** — unzip an Athena vocabulary download directly into the same directory `cdm database.connection string:` points at (`etc/etl-engine.d/cdm-out/` in the shipped config). The engine reads `CONCEPT.csv`, `CONCEPT_RELATIONSHIP.csv`, `VOCABULARY.csv`, etc. **in place**, alongside the CSV files it writes for the rest of the CDM. **No automation step required.**
- **RDBMS target (Paid tier)** — drop the Athena zip into `etc/etl-engine.d/vocab/` and let `automation.pre.populate vocabulary: once` load it into the database on the next run. See `automation`.

For licensed code sets (CPT-4 etc.), run Athena's unpacking scripts before either approach — see Troubleshooting → Athena zips with licensed code sets. 5. **Source data** — either CSV files in `etc/etl-engine.d/db/`, or a reachable RDBMS named in the `source database` block.

Run

```
$ cd etl-engine-X.Y.Z/bin
$ ./run-etl-engine.sh
```

The run script:

1. Clears any previous content from `logs/` (so each run starts with a clean log directory).
2. Starts the `etl-engine` container in the foreground.
3. Mounts `etc/etl-engine/` (read-only), `etc/etl-engine.d/` (read-write), and `logs/` (read-write) into the container. The read-only mount on `etc/etl-engine/` is intentional — `etl.conf` and the Rabbit-in-a-Hat file should never change during a run.

   The container is started with `--privileged`, `--pid=host`, and `--security-opt seccomp=unconfined`, plus a bind-mount of `/sys/kernel/debug`. These flags exist so performance-profiling tools (perf, eBPF) can attach to the engine when diagnosing slow runs. They are not load-bearing for an ordinary transformation; if your security policy disallows any of them, edit `bin/run-etl-engine.sh` and remove the offending flag.

   The engine runs in the foreground until it finishes. Press Ctrl-C to abort early — the container is stopped immediately, and your data directories are left as they were at that point.

What gets written

Inside `logs/` you will find:

- `etl-engine.log` — the engine's own log, controlled by `logging.log level` in `etl.conf`.
- One log per non-vocabulary CDM table — named after the CDM table itself — containing the parsed rule chains for that table and any rows that failed to transform.

If the engine stops with no output on the terminal, `etl-engine.log` is the first place to check. See Troubleshooting → Log Files for log-level guidance and what to look for.

Re-running

A re-run is just `./run-etl-engine.sh` again. The engine clears `logs/` at the start of each run, so the log directory only ever reflects the most recent execution. If you need to keep older logs, copy them out of `logs/` before the next run.

For larger ETLs split across multiple Rabbit-in-a-Hat files, see Folder Layout → Splitting an ETL.

3.1.4 Folder Layout

Unzipping `etl-engine-X.Y.Z.tar.gz` creates the directory structure below. The engine expects this layout — keep the names intact and the shipped scripts will work without configuration changes.

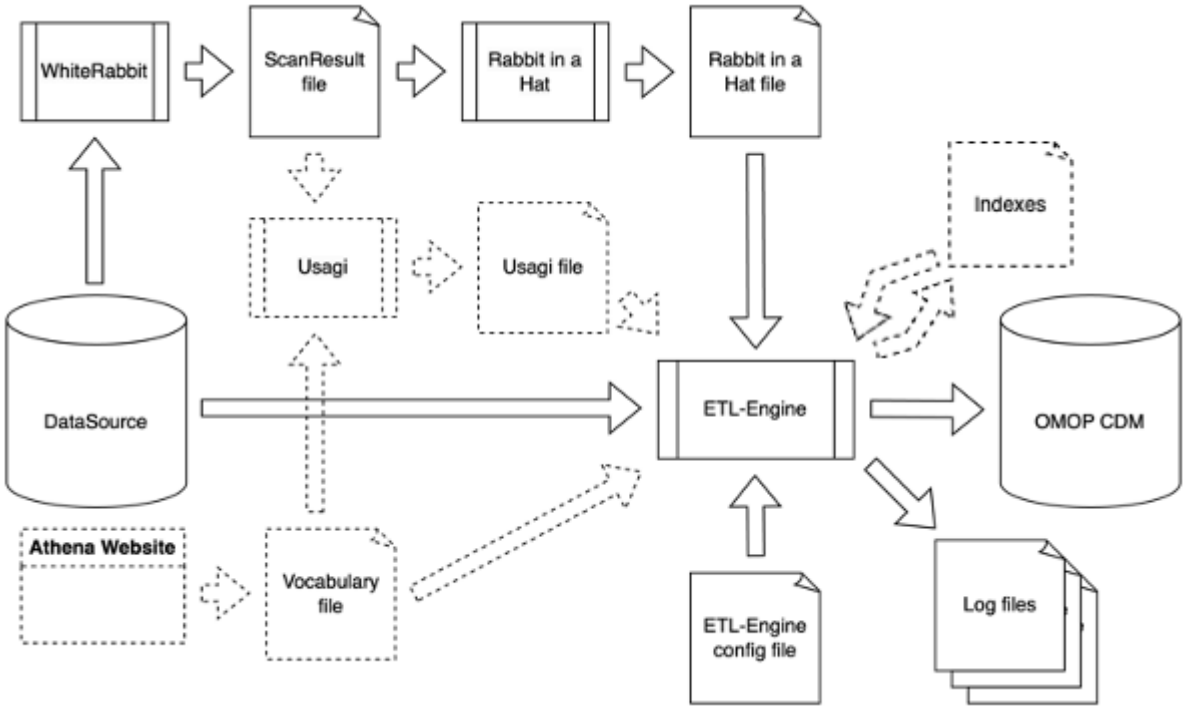
```
bin/install-etl-engine.sh    # Loads the Docker image
bin/run-etl-engine.sh        # Runs the engine container

data/etl-engine.docker       # Docker image bundled with this release

etc/etl-engine/etl.conf      # The engine configuration file
etc/etl-engine/              # Drop the Rabbit-in-a-Hat file here too

etc/etl-engine.d/db          # CSV source files (when source driver is "csv")
etc/etl-engine.d/usagi       # Usagi mapping files
etc/etl-engine.d/vocab       # Athena vocabulary zips (input to "populate
                             # vocabulary" automation)
etc/etl-engine.d/vocab.backup # Where processed Athena zips are moved
etc/etl-engine.d/indexes     # Persisted indexes and sequences from
                             # StoreIndexMap / StoreSequence

logs/                        # Log files written by the engine
                             # (cleared at the start of every run)

licenses/                    # Open-source library licenses

claude-skill/etl-engine.md   # Claude Code skill — pair with a local
                             # AI assistant for engine-specific help
```

The `claude-skill/` directory

The shipped `claude-skill/etl-engine.md` is a Claude Code skill — a single Markdown file packaging engine expertise (rule names, log shapes, configuration keys, debugging walkthroughs) for use by a local AI assistant on the system running the engine. See AI assistants for installation.

The rest of the documentation lives online at **docs.rook-it.com/etl-engine** — not bundled with the release, so the documentation can iterate faster than the engine binary. If you need an offline copy of the full site, mirror it locally with `wget --mirror` or equivalent.

Inputs into the engine



A typical run combines several artefacts:

- **Source database** — CSV files in `etc/etl-engine.d/db/`, or a connection string to a supported RDBMS (MariaDB / MySQL, PostgreSQL, or SQL Server) configured in `etl.conf`.
- **OMOP CDM vocabulary** — loaded into the destination database. The engine can load it from an Athena zip via the `automation.pre.populate vocabulary` setting.
- **Usagi files** — CSV files used by the `UsagiMap` rule, stored under `etc/etl-engine.d/usagi/`.
- **Rabbit-in-a-Hat file** — a gzipped JSON file describing the source-to-CDM mapping, stored alongside `etl.conf`.

Container view of the same paths

The run script mounts the host directories into the container as follows:

Host	Container
`etc/etl-engine/`	`/etc/etl-engine` (read-only)
`etc/etl-engine.d/`	`/etc/etl-engine.d` (read-write)
`logs/`	`/var/log/etl-engine` (read-write)

The shipped `etl.conf` references the **container** paths (`/etc/etl-engine.d/db/`, `/var/log/etl-engine/`, ...), which is what you'll see in error messages and log output.

Splitting an ETL across multiple Rabbit-in-a-Hat files

A single ETL does not need to live in one Rabbit-in-a-Hat file. When you split it, the engine can persist indexes and global sequences from one run so the next can pick them up — the typical case is when the first run populates the `person` table with new IDs, and a later run populates `visit_occurrence` and needs to map source person IDs back to the CDM `person_id` it already issued.

Use the `StoreIndexMap` and `StoreSequence` table rules in the first run; the same indexes are loaded automatically in subsequent runs from `etc/etl-engine.d/indexes/`.

3.1.5 Downloading a vocabulary from Athena

Rule chains in a Rabbit-in-a-Hat file that use `VocabMap`, `VocabSourceId`, or `UsagiMap` need a copy of the OMOP vocabulary loaded on the destination side. Vocabularies come from Athena, OHDSI's vocabulary download portal.

Rook IT can't distribute the vocabulary itself — every constituent vocabulary (UMLS, SNOMED CT, CPT-4, LOINC, RxNorm, …) has its own licence and Athena's account flow is where those terms get accepted. What we *can* do is tell you exactly which vocabularies your RiaH needs, and provide scripts that take it from there.

Step 1 — find out which vocabularies your RiaH needs

From the unpacked engine bundle:

```
./bin/check-vocabs.sh
```

Sample output for the bundled Synthmas RiaH:

```
OMOP vocabularies referenced by the RiaH:
  ✗  CVX                 MISSING — re-download from Athena with this vocab selected
  ✗  LOINC               MISSING — re-download from Athena with this vocab selected
  ✗  RxNorm              MISSING — re-download from Athena with this vocab selected
  ✗  SNOMED              MISSING — re-download from Athena with this vocab selected

Usagi files referenced by the RiaH:
  ✓  specialities.csv
  ✗  values-observation.csv  MISSING
  ...
```

The script scans every rule chain in your hat file for `VocabMap`, `VocabSourceId`, and `UsagiMap` references, and tells you which vocabularies (by Athena's short name) and which Usagi files the rules will look up at run time. Re-run it after each `setup-vocab.sh` to confirm the vocab you downloaded actually covers what the RiaH needs.

Step 2 — download from Athena

1. Open https://athena.ohdsi.org/ and **create an account** if you don't have one (free, but required — Athena needs an account to enforce the constituent vocabulary licences).
2. Click **Download** in the top navigation.
3. **Select the vocabularies** the previous step listed (e.g. `SNOMED`, `RxNorm`, `LOINC`, `CVX` for the bundled Synthmas demo). Athena's UI organises them roughly by Standard / Classification / Non-standard; the search box at the top filters quickly.
4. Click **Download** at the bottom of the selection list.
5. Athena prompts you to **accept the terms** for each licensed vocabulary in your selection (UMLS, SNOMED, CPT-4, …). Accept only those you have rights to use.
6. Athena queues the export and emails you a zip when it's ready — typically within a few minutes. The file is named like `vocabulary_download_v5_<timestamp>.zip`.

> **CPT-4 and other licensed code sets**
>
> If you selected CPT-4 (or another licensed-with-API code set), the download zip contains placeholders rather than real concept rows. See Troubleshooting → Athena zips with licensed code sets for the unpacking script process. You'll need a UMLS API key.

Step 3 — install the vocabulary

For a CSV CDM target (the Free-tier default):

```
./bin/setup-vocab.sh ~/Downloads/vocabulary_download_v5_*.zip
```

The script unpacks the CSV files into `etc/etl-engine.d/cdm-out/`, skipping any non-CSV artefacts (Athena's unpacking scripts, READMEs, etc.). The engine reads the vocab tables in place when it runs.

For an RDBMS CDM target (Paid tier), use the `automation.pre.populate` `vocabulary` setting in `etl.conf` instead — see `automation`.

Step 4 — verify

```
./bin/check-vocabs.sh
```

You should now see ✓ marks against every referenced vocabulary:

```
OMOP vocabularies referenced by the RiaH:
  ✓  CVX                 present
  ✓  LOINC               present
  ✓  RxNorm              present
  ✓  SNOMED              present
```

If anything is still ✗, re-download from Athena with the missing vocab selected, and re-run `setup-vocab.sh`.

Usagi files

The check script also lists Usagi files your RiaH expects. These are domain-specific source-string → concept-id mapping CSVs that you build yourself with the OHDSI Usagi tool — they aren't distributed either by Rook IT or by Athena.

If your RiaH references a Usagi file that doesn't exist yet:

1. Open Usagi (free, separate download from Athena).
2. Feed it the distinct source values you want to map, plus a vocabulary to map against.
3. Curate the suggested mappings.
4. Export to CSV, drop the file into `etc/etl-engine.d/usagi/`, named to match the `UsagiMap("…")` reference in the RiaH.
5. Re-run `./bin/check-vocabs.sh` to confirm all references resolve.

What the engine does without vocabulary

If you skip the vocabulary download entirely, the engine still runs — the vocab-lookup rules just leave their concept fields NULL (or fail the row, depending on whether you've supplied a default). The rest of the CDM populates normally. So a quick smoke test of the engine against Synthmas works without ever opening Athena; the vocabulary step is what turns a structural transformation into a clinically-meaningful one.

3.1.6 AI assistants

The engine ships with a **Claude Code skill** in `claude-skill/etl-engine.md` — a single markdown file that encodes the operational knowledge for authoring Rabbit-in-a-Hat mappings, configuring `etl.conf`, and debugging failed runs. Drop it into your Claude Code setup and you can ask detailed engine questions without the assistant needing the source or the hosted docs.

The same content is portable to other AI tools — see Other AI tools below.

Installing the Claude Code skill

Claude Code loads skills from one of two locations. Pick whichever suits your workflow:

USER SCOPE — AVAILABLE IN EVERY CLAUDE CODE SESSION

```
mkdir -p ~/.claude/skills
cp claude-skill/etl-engine.md ~/.claude/skills/
```

The skill is now available globally on this machine, regardless of which directory you start Claude Code in.

PROJECT SCOPE — COMMITTED ALONGSIDE YOUR ETL PROJECT

```
cd <your-etl-project>
mkdir -p .claude/skills
cp /path/to/claude-skill/etl-engine.md .claude/skills/
git add .claude/skills/etl-engine.md
git commit -m "Add ETL Engine Claude skill"
```

The skill is now available to anyone who clones the project. This is the recommended scope for production engagements where multiple people on the team use Claude Code on the same mappings.

USING THE SKILL

In any Claude Code session, invoke it with:

```
/etl-engine
```

...and ask whatever you need:

- "How do I map a SNOMED code to a concept id?"
- "What does `Errors break ETL of table` mean?"
- "Show me the pre-map + VocabMap pattern."
- "Why is my `visit_occurrence_id` NULL?"
- "What's the difference between `Map` with and without a default?"

The skill stays grounded in the documented behaviour of the engine — it will tell you when it doesn't know something rather than invent syntax.

Updating the skill

When you upgrade the engine to a new release, replace the skill file with the version from the new release's `claude-skill/` directory — the skill is regenerated each release to match what the engine actually does.

```
cp <new-release>/claude-skill/etl-engine.md ~/.claude/skills/
# or, for project scope:
cp <new-release>/claude-skill/etl-engine.md <your-project>/.claude/skills/
```

Other AI tools

The skill body — everything below the `---`-bracketed frontmatter at the top of `etl-engine.md` — is plain markdown. Any AI assistant that accepts a system prompt or rules file can use it.

Tool	Where to put it
Cursor	Save the body as `.cursorrules` in your project root.
GitHub Copilot	Save as `.github/copilot-instructions.md`.
Continue	Reference as a system message in the Continue config.
ChatGPT (Custom GPT)	Paste the body into the Custom GPT's instructions panel.
Gemini / Generic LLM	Provide as context in any conversation, or as the system prompt for a custom assistant.

Strip the frontmatter (`---`-bracketed block at the very top) before using the body outside Claude Code — that block is only meaningful to Claude Code's skill loader. The rest is tool-agnostic markdown.

> **Why two files would be cleaner than one**
>
> A future release may ship the skill body as a separate `.md` file without frontmatter, and the Claude-specific frontmatter as a thin wrapper that includes it. Until then: strip the frontmatter manually for non-Claude-Code tools.

Privacy and data flow

The skill file is a **static knowledge base** — installing it does not make Claude Code (or any other tool) phone home, and does not share your mappings, source data, or run logs with anyone. The skill just teaches Claude *about* the engine; what you ask Claude is up to you and is governed by Claude Code's own data policies, not by the skill or by Rook IT.

Building your own skill

If you have proprietary mapping conventions or in-house glossary terms you want a Claude skill to know about, you can either extend this skill or write a project-specific companion skill that lives alongside it. A skill is just a markdown file with a small frontmatter block — see Anthropic's Claude Code skill documentation for the format. Two skills can coexist; Claude reaches for whichever is most relevant to the question.

3.2 Architecture

3.2.1 Architecture

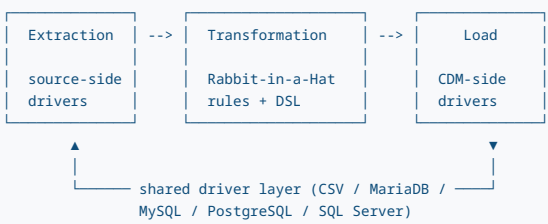
This section gives an overview of the ETL Engine's runtime architecture and its integration with the OHDSI tool family. Read this when you want to understand how the pieces fit together — installing and using the engine does not require this depth.

Pages

- Pipeline Overview — readers, transform manager, writer.
- Rabbit-in-a-Hat Integration — how the scan-report file becomes an executable transformation.
- Source & CDM Drivers — CSV, MariaDB / MySQL, PostgreSQL.

3.2.2 Pipeline overview

The ETL Engine is — exactly what its name says — an **E**xtract / **T**ransform / **L**oad pipeline. The three stages are clearly separated inside the engine, share a common driver layer, and run concurrently for as long as the dependency graph allows.



Extraction

Source-side drivers read records from the source database. Available drivers are:

- **CSV** — read a directory of CSV files. Drop them into `etc/etl-engine.d/db/`.
- **MariaDB / MySQL** — read from a live database via the standard connection string.
- **PostgreSQL** — same.
- **SQL Server** — read from Microsoft SQL Server via ODBC.

See Drivers for the connection-string formats.

Source data flows into the engine's typed value model — the four internal types are `INT`, `FLOAT`, `DATE`, and `STRING`. Database-specific column types (varchar, real, timestamp-with-zone, etc.) are mapped onto these four; the rest of the pipeline never has to care which RDBMS the data came from.

Transformation

The transformation stage is where the bulk of the engine's value lives. It takes the Rabbit-in-a-Hat file, the rule chains embedded in its `logic` fields — and in the `Comments` fields of tables and fields that have no `logic` field of their own — the Usagi mapping files, and the OMOP standard vocabulary, and produces transformed CDM rows.

Concretely, the engine:

1. Builds an internal model from the Rabbit-in-a-Hat file.
2. Resolves all DSL rule chains, validating types and parameters.
3. Builds a dependency graph among CDM tables (basic dependencies from Rabbit-in-a-Hat arrows, complex ones from `Join` / `LeftJoin`, and user-declared ones from `DependOn`).
4. Schedules tables in dependency order.
5. Dispatches source rows to a worker-thread pool, which runs the per-table rule chain on each row.

For the language and the rule catalog, see the DSL Guide and the Rule Reference.

Load

CDM-side drivers write transformed records to the destination database. The same driver set is available for the load side as for extraction — **CSV**, **MariaDB / MySQL**, **PostgreSQL**, and **SQL Server** — which means source and target choices are independent. Common combinations include:

Source	Target	When you'd use it
CSV	PostgreSQL	One-shot migrations from extracts.
PostgreSQL	PostgreSQL	Schema-to-schema CDM transformation in one DB.
SQL Server	PostgreSQL	Pulling from a SQL Server EHR into an OMOP CDM.
CSV	SQL Server	Loading bulk extracts into a SQL Server CDM.

Dependency model

The engine derives table dependencies from three sources:

- **Basic dependencies** — drawn explicitly in Rabbit-in-a-Hat as source-to-CDM table arrows.
- **Complex dependencies** — generated by `Join` / `LeftJoin` rules and by `GetCDMTableId`, which introduces an implicit dependency on the looked-up table.
- **User-declared dependencies** — the `DependOn` table rule, used when foreign-key relationships in the destination database are not inferable from the Rabbit-in-a-Hat model.

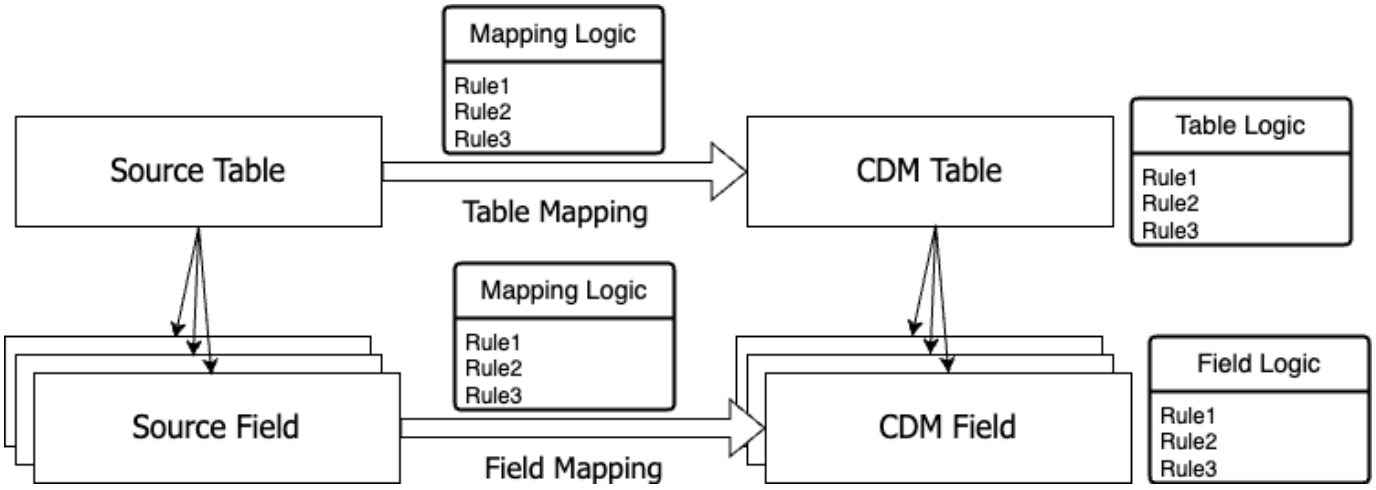
> ⚠ **Cyclic dependencies**
>
> `DependOn` can be used to construct a cycle, in which case no table is transformed and no data is loaded. The engine does not currently break cycles automatically — review your `DependOn` rules carefully.

What the engine does not do

- **It does not write SQL.** The destination driver issues bulk inserts; the rule language is value-oriented, not query-oriented.
- **It does not modify source data.** The source side is read-only.
- **It does not infer mappings.** Every CDM field that needs a value must be wired up in Rabbit-in-a-Hat — either as an arrow from a source field, a rule on the CDM field, or via a virtual field plus rule.

3.2.3 Rabbit-in-a-Hat integration

The ETL Engine consumes the JSON output of Rabbit-in-a-Hat, the OHDSI tool for designing source-to-CDM mappings. By reusing Rabbit-in-a-Hat as the front-end, the engine inherits a familiar visual modelling environment without requiring data owners to learn a new tool.



Artefacts the engine consumes

Rabbit-in-a-Hat provides six artefacts that the engine reads:

- **Source tables** — the rows on the left.
- **CDM tables** — the rows on the right.
- **Source-to-CDM table maps** — arrows from source to CDM tables.
- **Source fields** — fields inside source tables.
- **CDM fields** — fields inside CDM tables.
- **Source-to-CDM field maps** — arrows from source fields to CDM fields.

Each of those, plus the Comments fields on tables and fields, can carry rules from the engine's mapping language.

Where rules are placed in the model

Location	Prefix	Acts as
Source-to-CDM table arrow logic	*none*	Table-level rule on a single source table.
CDM table Comments	`logic:`	Table-level rule on the CDM table.
CDM table Comments	`field:`	Declares a virtual field.
Source-to-CDM field arrow logic	*none*	Field-to-field rule driven by one source field (its logic may also reference sibling source columns and same-row CDM fields — see below).
CDM field Comments	`logic:`	Field rule with all incoming maps as inputs.

What a field-to-field rule can reference

A source-to-CDM field arrow is *driven* by its one source field, but its logic is not limited to that single value. It can also read:

- **Other columns of the same source row.** Name them directly — e.g. a field arrow from `quantity` whose logic is `Add(:quantity, :refills)` also reads the `refills` column. The engine pulls every referenced column from the source row alongside the arrow's own field.
- **Other CDM fields of the same row being built.** A field's logic can key a lookup on a CDM value produced by another field's map. For example `GetCDMValue(@person:year_of_birth, [:person_id], [:person_id])` keys the lookup on *this* row's CDM `person_id`. The engine computes such a field after the fields it depends on; a circular dependency is rejected at configure time.

Together these make multi-key lookups expressible directly on a field arrow. A `cost_event_id`, say, can key `GetCDMValue` on a source code **and** the row's already-computed CDM `person_id` / `visit_occurrence_id` in one rule:

```
GetCDMValue(@procedure_occurrence:procedure_occurrence_id,
            [:person_id, :visit_occurrence_id, :procedure_source_value],
            [:person_id, :visit_occurrence_id, :code])
```

Each key value resolves independently — a source column (`code`) is read from the source row, a same-row CDM field (`person_id`) from the value computed for this row — so the two can be mixed freely.

What the engine does with the model

- **Internal mapping tables** — the engine builds its own internal representation, ignoring Rabbit-in-a-Hat's stem-table and completeness features.
- **Validation** — the engine checks that every table-to-table mapping is consistent (for example, all required CDM fields must be mapped) before loading.
- **Logic blocks** — anything in a Comments field starting with `logic:` is parsed as a rule chain. Anything starting with `field:` is interpreted as a virtual-field declaration.

3.2.4 Source & CDM drivers

The ETL Engine ships with five drivers. **Every driver works on either side** — source, CDM, or both. Source and CDM choices are independent: SQL Server in / PostgreSQL out is just as valid as CSV → CSV.

CSV

The CSV driver reads a directory of files. The connection string is the path to the directory:

```
source database {
  driver: "csv"
  connection string: "/etc/etl-engine.d/db/"
}
```

CSV is the simplest driver to set up and is a good choice for one-shot migrations and the initial development cycle. Each `.csv` file in the directory becomes a source table; on the target side, each CDM table is written to its own CSV file.

On the target side the CSV driver runs the **full CDM lifecycle**, the same as the RDBMS drivers: `create omop cdm` pre-creates each CDM file with the canonical OMOP header, `clean omop data` resets each file back to that header (the file-level equivalent of SQL `TRUNCATE`), and a single CDM table may be written by **several** `etl_map` **steps** that each append their own columns. A `CSV → CSV` run therefore behaves like `CSV → PostgreSQL` apart from where the rows land — useful for testing a mapping or building a portable extract with the exact CDM shape the database target would produce.

Date columns are written in **ISO 8601**: a `DATE` renders as `yyyy-mm-dd` and a `DATETIME` as `yyyy-mm-dd HH:MM:SS` — for example `2013-06-24` and `2013-06-24 06:20:52`. (Source dates in other formats are parsed with `AsDate`; the ISO form is only the output rendering.)

The derived-table automations — `observation_period`, `condition_era`, `drug_era` and `dose_era` — run against a CSV target too (`drug_era` and `dose_era` read the Athena vocabulary CSVs directly), and so does `add preceding visit ids` — on a CSV target it rewrites the produced `visit_occurrence` / `visit_detail` files in place. No post automation requires a database target. See `automation` → `post` block.

MariaDB / MySQL

The MariaDB driver speaks the MariaDB and MySQL wire protocol and uses the key-value connection string described in `cdm database`.

```
cdm database {
  driver: "mariadb"
  connection string: "user=omop dbname=omop_cdm host=db.example.com password=secret"
}
```

MariaDB does not support schemas, so there is no `options` key for setting a search path.

PostgreSQL (native)

The native PostgreSQL driver uses the standard `libpq` client and the same key-value connection-string format. It supports an `options` key, forwarded to the PostgreSQL client — typically `--search_path=` to select a CDM schema:

```
cdm database {
  driver: "postgresql"
  connection string: "user=postgres host=db.example.com port=5432 dbname=postgres options='--search_path=omop_cdm' password=secret"
}
```

PostgreSQL over ODBC (`psql-odbc`)

The same PostgreSQL backend, accessed through ODBC. Use this when your environment standardises on ODBC for all RDBMS connectivity, or when you need an ODBC-only driver path through a corporate ODBC gateway. Connection-string format follows ODBC conventions (semicolon-separated key/value pairs):

```
cdm database {
  driver: "psql-odbc"
  connection string: "Driver={PostgreSQL Unicode};Server=db.example.com;Port=5432;Database=omop;Uid=etl;Pwd=secret"
}
```

The shipped container bundles a PostgreSQL ODBC driver, so no host-side ODBC configuration is required when running the engine inside the container.

For most deployments the **native PostgreSQL driver** is the better choice — it's faster (no ODBC round-trip), handles `options` in the expected `libpq` format, and is the path most heavily exercised by the test suite.

SQL Server

The SQL Server driver also speaks ODBC. The connection string uses ODBC conventions:

```
cdm database {
  driver: "sqlserver"
  connection string: "Driver={ODBC Driver 18 for SQL Server};Server=sqlserver.example.com,
1433;Database=omop;UID=etl;PWD=secret;TrustServerCertificate=yes"
}
```

The shipped Docker image bundles Microsoft's ODBC Driver 18 for SQL Server, so no host-side ODBC configuration is required when running the engine inside the container.

Unicode source columns (`nvarchar` / `nchar`) are fully supported — the engine reads them via a `VARCHAR` cast, so tables whose text columns are declared Unicode load the same as non-Unicode ones. Identifiers are quoted with SQL Server's `[bracket]` syntax, so reserved words and mixed-case table/column names work on both the source and CDM sides.

For more on connection-string formats, consult the relevant database vendor's documentation.

Choosing a driver combination

There is no required pairing between the source and CDM drivers. Some common patterns:

- **CSV → PostgreSQL** — source data in flat-file extracts, CDM in PostgreSQL.
- **PostgreSQL → PostgreSQL** — schema-to-schema CDM transformation inside a single database.
- **SQL Server → PostgreSQL** — pulling from a SQL Server EHR into an OMOP CDM hosted in PostgreSQL.
- **CSV → SQL Server** — bulk-loading from extracts into a SQL Server CDM.
- **CSV → CSV** — testing a mapping or building a portable extract.

3.3 DSL Guide

3.3.1 DSL Guide

The ETL Engine ships its own mapping language for the cases that Rabbit-in-a-Hat alone cannot express — joins, conditional chains, vocabulary lookups, multi-sized results, and so on. This guide covers the language in depth.

Pages

- Mapping Language Syntax — formal grammar and worked examples.
- Pipelines and Chaining — the `=>` operator for explicit rule chains and named field bindings.
- Conditional Chaining — the `||` operator for fallbacks between rules.
- Multi-Sized Results — `[]` and `[N]` suffixes, `LoopOver`, and how multi-result rules fork execution paths.
- Arrays and New Mode — the `ARRAY` value type, `new mode` configuration, and array-aware rules.
- ForEach and Blocks — the `|` ForEach operator and block syntax.
- Virtual Fields — defining and using virtual fields on CDM tables.

Where rules live in Rabbit-in-a-Hat

The engine reads rules from four places in the Rabbit-in-a-Hat model:

1. **Source-to-CDM table maps** — rules placed on the arrow from a source table to a CDM table act as table-level rules.
2. **CDM tables** — rules placed on a CDM table itself, in the `Comments` field, with the prefix `logic:`.
3. **Source-to-CDM field maps** — rules on the arrow from a source field to a CDM field; default behaviour for field-to-field mappings.
4. **CDM fields** — rules placed on a CDM field, in the `Comments` field, with the prefix `logic:` (or `field:` to declare a virtual field).

The full list of rules per category is in the Rule Reference.

3.3.2 Mapping Language Syntax

This page documents the formal grammar of the mapping language and the semantics of each production.

Grammar

Each row below names a leaf in the syntax tree (left) and its expansion (right), written as a mix of leaf names and regular expressions. The most general production — `RuleChain` — is at the top.

Leaf	Expansion
`RuleChain`	`Rule_Definition+`
`Rule_Definition`	`[Input_Field_Filter =>]? Rule [|| Conditional_Rule]* [=> Output_Fields_List]?`
`Input_Field_Filter`	`Field_List`
`Conditional_Rule`	`Rule [\[ [Integer]? \]]?`
`Rule`	`Rule_Name\(Rule_Parameters\)`
`Output_Fields_List`	`Field_List`
`Field_list`	`Field [, Field_list]*`
`Rule_Name`	`[A-Z][A-Za-z]+`
`Rule_Parameters`	`Parameter [, Rule_Parameters]*`
`Parameter`	`Keyword_Parameter | Table_Field | Table | Field | String | Date | Integer | Float | Regex | NULL`
`Keyword_Parameter`	`[a-z][a-z0-9._]= ( Table_Field | Table | Field | String | Date | Integer | Float | Regex | NULL )`
`Table_Field`	`@[A-Za-z][A-Za-z0-9._]+ :[A-Za-z][A-Za-z0-9._]+`
`Table`	`@[A-Za-z][A-Za-z0-9._]+`
`Field`	`:[A-Za-z][A-Za-z0-9._]+`
`String`	`".*"`
`Date`	`'.*'`
`Integer`	`[0-9]+`
`Float`	`[0-9]+\.[0-9]+`
`Regex`	`/.*/`

Semantics

`RuleChain`

At the start of execution the environment is set up:

- In a **field-to-field map** the environment contains exactly one input field (the source field on the arrow's tail) and one output field (the destination field on the arrow's head). The output field is unnamed and doesn't need a specific name. By the end of execution exactly one field must be present — multiple values cause an error and the table is not mapped.
- In a **field rule** (rules on a CDM field directly) the environment contains every field that maps into that CDM field — possibly zero. By the end of execution exactly one field must remain.

Rules in a `RuleChain` execute top-down: changes to a field or to the environment affect every subsequent rule.

`Rule_Definition`

A rule definition is a set of inputs (explicit or implicit), a rule (or conditional rule chain) to execute, and a list of outputs (explicit or implicit).

`Input_Field_Filter`

A filter placed in front of a rule restricts the fields passed to it. Every field referenced in the filter must already exist in the environment — either as a source field or via an `Output_Fields_List` from a preceding rule.

When an input filter is present but no `Output_Fields_List` is, behaviour depends on the rule type:

- **Mapping rules** update the fields named in the filter.
- **Aggregation rules** delete those fields and create a new output field.

`Conditional_Rule`

Rules can be joined into a conditional chain with `||`: if the first rule fails to produce a mapping, the second is tried, and so on.

- **Mapping rules** can only be conditionally chained with other mapping rules.
- **Aggregating rules** can only be conditionally chained with other aggregating rules.
- Not all rules support being conditionally chained at all — see the individual rule pages.

`Rule`

Rules behave differently depending on family:

- **Mapping rules** execute one-to-one over input fields. When applied to a field rule (not a field-to-field map), the rule runs on every input field unless an input filter is present. The output list may name only a single field, so a rule that names its output must take exactly one input — see `Output_Fields_List`.
- **Aggregating rules** always have a single output, but the number of inputs is unbounded. An input filter works exactly as for mapping rules, but the output list may name only a single field.

Rules can also produce **multi-sized results** — see Multi-Sized Results.

`Output_Fields_List`

An output list may name **exactly one field**, whatever the rule family. A new name declared here becomes a virtual field.

Behaviour depends on rule family:

- **Mapping rules** write the result to the named field. Because the list may name only one field, a rule that names its output must take exactly one input — `:gender => AsString() => :result` is valid, `:gender, :race => AsString() => :g, :r` is not.
- **Aggregating rules** delete every input field except the one mentioned in the output list. If the output list names a field that already exists in the input, that field is preserved with the new value.

> ⚠ **One output name per rule**
>
> Naming one output per input — `:f1, :f2 => AsString() => :s1, :s2` — is **not supported**, and neither is naming fewer outputs than inputs. Both are rejected at startup and the table fails to map; see Only one output name can be specified for a mapping rule.
>
> To transform several fields *without* renaming them, omit the output list — a mapping rule with an input filter and no output list updates each named field in place:
>
> ```
> :f1, :f2 => AsString()
> ```
>
> To rename as well, use one rule per field:
>
> ```
> :f1 => AsString() => :s1
> :f2 => AsString() => :s2
> ```

Worked examples

STATIC VALUE MAPPING

```
Map([
  ["M", 8507],
  ["F", 8532]
])
```

A mapping rule on a field-to-field map. The input is expected to be `"M"` or `"F"`; the rule replaces the source value with the matching code.

CONDITIONAL CHAINING FOR THE SAME EFFECT

```
Map([["M", 8507]]) || Map([["F", 8532]])
```

If the first `Map` does not match, the second is tried. The end result is identical to the single map above.

EXPLICIT INPUT/OUTPUT FILTERS

```
:gender => Map([
  ["M", 8507],
  ["F", 8532]
]) => :result

:result => AssertNot(8532)
Use(:gender)
```

The first chain stores the map result into a new virtual field `:result`. `AssertNot` then runs against `:result` only — `:gender` is not asserted on. Finally `Use(:gender)` declares `:gender` as the output field for the chain and discards `:result`. The same effect could be achieved by simply writing `AssertNot("F")`.

> **Use filters sparingly**
>
> Input filters and output lists are expensive at runtime. Reach for them only when there is no simpler way to express the chain.

AGGREGATION ON A CDM FIELD

```
logic: Selector(:int1, "=", :int2, "Equal", "Unequal")
```

Two source fields (`int1` , `int2`) are mapped into the CDM field. If they are equal, the string `"Equal"` is stored in a virtual field called `Selector`; otherwise `"Unequal"`. All input fields, used or not, are deleted by the aggregation.

COMBINING SEVERAL AGGREGATIONS

```
logic: :int1, :int4 => Min(:int1, :int4) => :min
logic: :int2, :int3 => Max(:int2, :int3) => :max
logic: Selector(:min, "<", :max, :min, :max)
```

The first two lines find `:min` and `:max`, leaving the chain with five fields (`:int1` … `:int4` , `:min` , `:max`). The final `Selector` chooses the lower of the two and stores it in a virtual field `Selector` — the only remaining field.

MULTI-SIZED RESULTS

```
UsagiMap("usagi.csv")[]
```

The `[]` suffix turns the rule into a multi-sized rule: each input record yields one output record per match. A bounded form `[3]` requires exactly three matches and skips the row otherwise.

3.3.3 Pipelines and chaining

> **New in 1.4**
>
> The `=>` pipeline operator and explicit named bindings are part of the new parser. To use them, ensure the engine is built from a 1.4 or later release.

The pipeline operator `=>` chains rules explicitly. It replaces the implicit field-flow model of the older syntax for cases where readability matters.

Single-line pipeline

```
:source_field => AsString() => PrePad(5, "0") => :zip
```

Read left-to-right: take `:source_field`, cast to string, pad with leading zeros to length 5, store as `:zip`.

What `=>` actually does

`=>` does **not** pipe a value from one rule into the next. It is shorthand for a series of rules run one after another over the same set of fields — the same fields the input filter names. Each step's effect on that set is what carries forward, and that depends on the rule's family:

- A **mapping** rule updates the fields in place, so the filter still names them for the next step.
- An **aggregating** rule consumes those fields and produces a single result, so the steps after it work on that result instead.

That is why the two families read so differently in a pipeline:

```
:f1, :f2 => AsInt() => Add() => :sum
```

`AsInt` is a mapping rule, so it converts **both** `:f1` and `:f2` and leaves them in place. `Add` is aggregating, so it consumes both and yields one value: `:sum` is `:f1 + :f2`. The engine expands it to:

```
:f1, :f2 => AsInt()
:f1, :f2 => Add() => :sum
```

The trailing output list names the aggregation's result, so a mapping rule after an aggregation updates that result in place:

```
:f1, :f2 => Trim() => Concat() => AsString() => :joined
  becomes
:f1, :f2 => Trim()
:f1, :f2 => Concat() => :joined
:joined  => AsString()
```

With several aggregations, each one writes the same name — so the value stays reachable the whole way down:

```
GetCDMFieldValue(:year_of_birth)[] => Size() => :yb
  becomes
GetCDMFieldValue(:year_of_birth)[] => :yb    // :yb is now an ARRAY
:yb => Size() => :yb                         // :yb is now its INT length
```

> **An aggregating step needs its inputs named**
>
> Because an aggregation consumes the fields the filter names, anything after it reads the aggregation's result — not the original fields. If the pipeline has **no** input filter, an aggregating rule with no parameters falls back to every field currently in scope, which is rarely what a long chain wants. Either name the filter (`:f1, :f2 => Add()`) or name the rule's operands (`Add(:f1, :f2)`).
>
> Engines up to and including 1.4.1 applied the filter to the *first* step only, so an aggregating rule after a `=>` silently aggregated every field in scope and returned a plausible wrong number instead of erroring. If you are reading a chain written against one of those, check it.

A rule that names fields must come first

A rule that takes **no input of its own** — `SetValue` , `GetCDMFieldValue` , and any rule like them — produces a value without reading the chain's fields, so it **does not carry the chain's existing bindings forward**. Every field the chain started with is gone for the steps after it. A later rule can still work on the value that rule produced, but it can no longer name the original fields:

```
SetValue(0) => Add(:start, :interval)     # error: :start is no longer in scope
Add(:start, :interval) => :end            # correct: Add sees the fields, runs first
```

Both `:start` and `:interval` are real bindings (they arrow into this CDM field), but `SetValue` runs first and drops them, so by the time `Add` names them they cannot be resolved. This is a build-time error — the table fails to load rather than silently producing no rows:

```
named input out of scope: `:start' is referenced here but a preceding rule
already dropped it from the chain. A rule that takes no input of its own
(SetValue, GetCDMFieldValue, …) does not carry the chain's existing field
bindings forward, so any rule that references those fields by name must come
before it, not after. Put the field-referencing rule first.
```

Mapping rules are fine after such a rule, because they work on the produced value in place rather than naming other fields — `SetValue(0) => AsString() => PrePad(5, "0") => :code` is valid. The rule is only about **naming** fields that were dropped. (Contrast a field that is simply *not mapped* into the CDM field at all: that gives "doesn't reference a source field mapped to the CDM field" instead.)

Multi-step pipelines

A pipeline with several aggregating steps composes naturally:

```
:int1, :int4 => Min() => :min
:int2, :int3 => Max() => :max
Selector(:min, "<", :max, :min, :max)
```

Why pipelines are preferred over input/output filters

The pre-pipeline syntax used input filters and output lists for the same job. Pipelines compile to the same execution but read more naturally and allow intermediate values to be reused without spelling out every filter.

See also: Conditional Chaining, ForEach and Blocks, Mapping Language Syntax.

3.3.4 Conditional chaining

The `||` operator joins rules into a fallback chain: if the first rule fails to produce a mapping, the second rule runs against the same input, and so on. This is invaluable when the source data has multiple plausible mappings and the engine should try them in priority order.

Pre-map plus vocabulary lookup

A common pattern: a `Map` supplies hand-curated codes for the values that the vocabulary cannot resolve, and a `VocabMap` handles the rest:

```
Map([
  ["legacy_code_A", "ICD10_A12.3"],
  ["legacy_code_B", "ICD10_B45.6"]
]) || VocabMap("ICD10")
```

If `Map` finds the input value, its replacement flows on. Otherwise `VocabMap` runs against the original input.

Two parallel maps with a `Selector`

The pre-map approach above only works if the chosen vocabulary actually has a code you can pre-map to. When that is not the case, the alternative is to run two independent rules into separate variables and combine with `Selector` — but this approach is **slower and harder to read** than `||` and should be reserved for genuinely independent mappings.

Family restrictions

- Mapping rules can only be `||`-chained with other **mapping** rules.
- Aggregating rules can only be `||`-chained with other **aggregating** rules.
- Not every rule supports `||` chaining. Each rule's reference page declares whether it does (look for `supports ||` in the meta line).

Multi-step chains

Conditional chains can have any number of rules — the engine walks them in order until one succeeds:

```
Map([...]) || UsagiMap("file.csv") || Map([..., default_code])
```

The final entry can be a `Map` with a default value as a catch-all that guarantees a successful mapping.

Array-typed fallbacks (new mode)

When **any** branch of a fallback carries a multi-sized-result suffix (`[]` / `[N]`) — for example a `UsagiMap` or `VocabMap` that can return several codes — the whole field becomes `ARRAY`-typed, and the fallback yields an `ARRAY` on **every** row regardless of which branch wins:

```
{ UsagiMap("codes.csv")[] } || { Map([...], 0) }
```

An array-producing branch contributes its array unchanged; a **scalar** winning branch (here the `Map` sentinel) has its single value wrapped into a **one-element array** (`0` → `[0]`). The field's value type is therefore static and consistent — *"there is a `[]` in the chain, so this field is an array"* — and downstream array operations (`Unique`, `At`, `RemoveAt`, `Spawn`, ...) compose uniformly on every row. Without this, a scalar-winning row would have no array to operate on and the array op would reject the row.

When the field is written straight to a column under a table `LoopOver`, a one-element array flattens to exactly the same single cell a bare scalar produces — so the written output is unchanged; the array typing only matters when you keep operating on the result. (Scalar-only fallbacks — no `[]` in any branch — are unaffected and stay scalar.)

3.3.5 Multi-sized results

Some rules can produce **more than one output value per input record**. This fans the execution out: every downstream rule in the chain runs once per result, and the source row produces multiple CDM rows.

The `[]` and `[N]` suffixes

Suffix	Meaning
`[]`	Unbounded — produce as many output rows as the rule returns matches.
`[N]`	Exactly `N` results required. If fewer are available, the row is skipped and an error is logged.

```
UsagiMap("file.csv")[]    # variable number of results
UsagiMap("file.csv")[3]   # exactly 3 results required
```

`LoopOver` controls the cartesian product

When several fields produce multi-sized results, the engine needs to know how to combine them. The `LoopOver` table rule declares this at the CDM-table level:

- Fields in the **same array** are zipped — the i-th result of one is paired with the i-th result of the others.
- Fields in **different arrays** form a cartesian product.

```
LoopOver([:field_one])
LoopOver([:field_one, :field_two])
LoopOver([:field_one, :field_two], [:cartesian_field_three])
```

Any mapping with a multi-sized result must declare `LoopOver`, even if only one field is multi-sized.

Forking and converging execution paths

A multi-sized rule **forks** the execution path. Successive multi-sized rules multiply forks, so the engine may end up tracking many parallel `(var1, var2, …)` tuples per source row. Subsequent aggregations such as `Unique` can be used as a **convergence point**: every fork's value is collected, deduplicated, and the chain refurnishes one fork per unique value.

> ⚠ **Don't put `Unique` inside a conditional branch**
>
> If some forks reach `Unique` and others don't, the post-`Unique` state is inconsistent — some forks see the converged value, others the original variables. This is rarely what you want.

3.3.6 Arrays and new mode

New in 1.4

The `ARRAY` value type and array-aware rules are part of the new execution mode. Enable with `new mode: "true"` in the `tool configuration` block.

The new mode introduces a first-class `ARRAY` value type, eliminating most multi-sized-result complexity in favour of explicit array values and rules that operate on them.

Enabling new mode

```
tool configuration {
  new mode: "true"
}
```

When new mode is enabled:

- `Use(:f1, :f2, :f3)[N]` collects fields into a single `ARRAY` value rather than forking the chain.
- `UsagiMap(...)[]` returns an `ARRAY` of all matches.
- `LoopOver` automatically expands any `ARRAY` field into row-level fan-out at the CDM-table boundary.

Array-aware rules

Rule	Purpose
`Spawn`	Turn an `ARRAY` into individual rows.
`CartesianOf`	Cartesian product of N input arrays.
`Unique`	Deduplicate elements across input arrays.
`Without`	Set difference of two arrays.
`Size`	Length of an array as `INT`.
`RemoveItem` *(upcoming)*	Drop all elements equal to a value.
`RemoveIdx` *(upcoming)*	Drop the element at a 0-based index.

`RemoveItem` and `RemoveIdx` ship with a forthcoming release that introduces additional element-wise array helpers; until that lands, the ForEach operator combined with a comparison rule covers the same use cases.

See ForEach and Blocks for the `|` operator, which applies a mapping rule per array element.

3.3.7 ForEach and blocks

> **New in 1.4**
>
> The `|` ForEach operator and `{}` block syntax are part of the new parser.

ForEach (`|`)

The `|` operator applies a mapping rule to every element of an `ARRAY`, returning a new `ARRAY`:

```
:codes => RemoveItem(".") | AsInt() => :int_codes
```

Read left-to-right: take `:codes` (an array), drop any `"."` elements, then cast each remaining element to `INT`, producing a new array `:int_codes`.

`|` and `=>` compose freely:

```
Use(:f1, :f2, :f3, :f4)[4]                  # collect 4 fields into an ARRAY
  => RemoveItem(".")                        # drop placeholder elements
  | AsInt()                                 # cast each remaining element
  => :year_of_birth                         # store the resulting array
```

Blocks (`{}`)

A block groups multiple rule chains that share a scope:

```
{
  :raw => UsagiMap("a.csv") => :ua
  :raw => VocabMap("ICD10") => :vm
  Selector(:ua, "!=", NULL, :ua, :vm)
}
```

All chains inside the block see the same field environment, and the block's final value is the chain's overall output.

See Arrays and New Mode for the underlying `ARRAY` type and the array-aware rules.

3.3.8 Virtual fields

Virtual fields are CDM-table fields that **only exist inside the engine** — they never reach the destination database. They are used as scratch space for normalisation, indexing, or holding intermediate values for joins.

Declaring a virtual field

Add a `field:` line to the **Comments** field of the CDM table in Rabbit-in-a-Hat:

```
field: visit_occurrence_source_value
```

The text after `field:` becomes the name of the virtual field. The field is then visible to every rule running on that CDM table — both field-to-field maps and CDM-field rules.

The example above adds a `visit_occurrence_source_value` field to `visit_occurrence` — a common requirement on CDM 5.4, where `visit_occurrence` lacks a `*_source_value` field for its primary key.

Writing to a virtual field from a field-to-field map

Use `SaveToVirtualField` on a transition that targets an otherwise-unrelated field; the rule redirects the value into the named virtual field:

```
SaveToVirtualField(:care_site_map)
```

This is necessary because Rabbit-in-a-Hat does not visualise virtual fields, so there is no transition arrow you could place a normal rule on.

Writing from rule chain output

Any rule chain's `Output_Fields_List` (the part after the trailing `=>`) can declare a new field. A field declared this way becomes a virtual field and remains in scope for subsequent rules in the same chain. A rule may declare only one output field, so use one rule per field you want to create:

```
:gender => Map([["M", 8507], ["F", 8532]]) => :result
:result => AssertNot(8532)
Use(:gender)
```

Here `:result` is a virtual field that lives only for the duration of the chain.

3.4 Troubleshooting

3.4.1 Troubleshooting

The ETL Engine is not chatty on stdout. Almost every diagnostic signal arrives through the log files the engine writes during a run.

When something goes wrong

The recommended workflow when an ETL run fails:

1. **Check that the engine started at all.** The terminal output of `run-etl-engine.sh` shows the version banner and any configuration-loading errors. If you see one of those, the run never reached the mapping stage — see Common Errors.
2. **Open `etl-engine.log`.** This is the engine's own log. Errors at `ERROR` level here usually indicate an engine-level problem; errors at lower levels are typically about the Rabbit-in-a-Hat configuration. If the engine stopped silently with no terminal output, this is the place to look.
3. **Open the per-table log for the table that failed.** Each non-vocabulary CDM table has its own log under `logs/`, named after the CDM table. The trailing `[critical] Errors break ETL of table` line tells you the table failed to load; the actual cause is one or more earlier `[error]` lines in the same file.
4. **Raise the log level if you need more detail.** Set `logging.log level` to `DEBUG` or `TRACE` in `etl.conf` and re-run. See Log Files.
5. **Use `TraceWhen`** for surgical debugging of one specific source row that's misbehaving.

Pages

- Log Files — what each log contains and how to read them.
- Common Errors — frequently seen messages and how to resolve them.
- Performance Tuning — knobs that affect throughput.
- TraceWhen — per-record verbose logging for targeted debugging.

Getting help

Issues that turn out to be engine bugs (rather than mapping issues) should be reported to your Rook IT contact, with:

- The engine version (printed at the start of `etl-engine.log`).
- A copy of `etl.conf`.
- The relevant table's log file.
- Where possible, a minimal Rabbit-in-a-Hat reproduction.

3.4.2 Log files

When the engine starts, it accepts at most one argument: a path to a local configuration file. Without one it falls back to `/etc/etl-engine`. If the configuration file fails to open or parse, the engine prints to stderr:

```
$ ./etl-engine etl.conf
ETL-Engine version: 1.1.0 Build d612c89
©2023 Rook-IT ApS

  Couldn't open file: etl.conf (-2)

Couldn't initialize etl-engine
```

or:

```
$ ./etl-engine etl.conf
ETL-Engine version: 1.1.0 Build d612c89
©2023 Rook-IT ApS

Missing source connection string
```

Once the configuration loads and the log directory is writable, **all subsequent communication is through the log files**.

`etl-engine.log`

The engine's main log. Its verbosity is set by `logging.log level` in `etl.conf`. When the engine stops with no output on the terminal, this is the first place to look.

Log levels:

- **TRACE** — everything: reading the Rabbit-in-a-Hat file, building the internal mapping, optimisation choices, join planning.
- **DEBUG** — same scope minus internal details. Enough to pinpoint where an error originates.
- **INFO** — stage transitions only (loading, cleaning, mapping).
- **WARN** — recoverable surprises, such as a failed rule-chain parse or a cascading mapping failure.
- **ERROR** — fatal errors only. A single line stating the cause of death.

Errors raised inside `etl-engine.log` during the configuration phase typically indicate an engine-level bug, not a user mapping problem.

Per-table logs

Once the engine begins to configure the transformation, one log file per non-vocabulary CDM table appears in the log directory, named after the table (`person.log`, `condition_occurrence.log`, ...). These logs always run at INFO level — there is no way to silence them.

Each table log contains:

- The parsed rule chains for the table — useful when debugging.
- Any rows that failed to transform, with the cause.
- A trailing `[critical]` line if the table failed entirely:

```
[2023-05-23 15:26:38.962] [device_exposure] [critical] Errors break ETL of table
```

That line means the table was not mapped, and any other CDM tables that depend on it will also fail.

The good news: errors in per-table logs are almost always caused by a bad Rabbit-in-a-Hat configuration, not by an engine bug. Fix the mapping in Rabbit-in-a-Hat, regenerate the file, and re-run.

3.4.3 Common errors

This page collects the messages most often seen during a run, what they mean, and how to fix them. Errors are grouped by where they appear: terminal output (the engine wouldn't start) versus per-table log files (the engine ran, but a table failed to load).

Engine startup errors

These appear on the terminal when `run-etl-engine.sh` fails before any log file is written.

`Couldn't open file: etl.conf (-2)`

The engine could not find or read `etl.conf`. Check that `etc/etl-engine/etl.conf` exists and is readable by the user (or container) running the engine. The shipped `run-etl-engine.sh` mounts `etc/etl-engine/` read-only into the container at `/etc/etl-engine`, so the file must exist on the host before you start the run.

`Missing source connection string`

The `source database` block in `etl.conf` has no `connection string` key. See the `source database` reference.

`Missing destination connection string`

Same as above for the `cdm database` block. See the `cdm database` reference.

`Couldn't initialize etl-engine`

Generic catch-all printed when the engine cannot bring itself up. The preceding line tells you which specific component failed — connection to the source database, parsing of the hat file, license validation, or similar.

Per-table errors

These appear in the per-table log file under `logs/<cdm-table>.log` and almost always indicate something the user can fix in the Rabbit-in-a-Hat configuration.

`Field 'X' is non-nullable and not mapped or defined`

```
note.log:[2023-05-23 15:26:38.950] [note] [error] Field 'note_id'
is non-nullable and not mapped or defined
```

The CDM table has required fields with no incoming mapping. **This is not always a problem** — if you do not intend to populate this table at all, the message is benign and the table simply is not loaded.

To populate it: map the listed fields in Rabbit-in-a-Hat. To leave it empty intentionally: ignore the message.

`Semantic error: The declared rule 'X' is undefined`

```
[device_exposure] [error] Semantic error: The declared rule `Dates' is undefined.
```

A rule chain references a rule the engine does not know. Typical causes:

- A typo (`Dates` instead of `Days`).
- An older Rabbit-in-a-Hat file that uses a rule name that has since been renamed.
- A version mismatch between the configured engine and the Rabbit-in-a-Hat file (the rule does exist, just not in this engine version).

Cross-reference the Rule Reference for canonical names. Fix the rule name in Rabbit-in-a-Hat and re-run.

`Errors break ETL of table`

```
[device_exposure] [critical] Errors break ETL of table
```

Always the last line of a per-table log when the transformation fails. The real cause is one of the earlier `[error]` lines in the same file — search backwards. Any CDM tables that depend on the broken table will also fail.

`The field 'X' is required, but it is not specified` (INSERTRECORD)

```
[cdm_source] [error] InsertRecord: The `cdm_release_date' is required,
but it is not specified.
```

An `InsertRecord` call omits a non-nullable field. Add the field as a `field=value` keyword parameter, or change the database schema so the field is nullable.

`The field input to <Rule>: ':X' doesn't reference a source field`

The named field does not exist on the source table being mapped. This typically means a rule references `:patient_id` while the source table only has `:id`. Either rename the source field in Rabbit-in-a-Hat to match, or adjust the rule to reference the correct field name.

`Only one output name can be specified for a mapping rule`

```
[person] [error] AsString: Only one output name can be specified for a
mapping rule - got 2 (:s1 and :s2). Naming one output per input isn't
supported; use a separate rule per output name.
```

An output list names more than one field, as in `:f1, :f2 => AsString() => :s1, :s2`. One output name per rule is the limit, for every rule family.

To transform several fields *without* renaming them, drop the output list — a mapping rule with an input filter and no output list updates each named field in place (`:f1, :f2 => AsString()`). To rename as well, use one rule per field:

```
:f1 => AsString() => :s1
:f2 => AsString() => :s2
```

The closely related `When saving mapping output into variables, all input variables must be specifically mapped ... got 2 input values and 1 output value` is the same limit seen from the other side: an output list that names *fewer* fields than the rule takes as input. The fix is the same.

`The CDM table must exist` (DEPENDON)

```
[X] [error] The referenced CDM table must exist.
```

A `DependOn` rule references a CDM table that is not part of this run — perhaps because of `cdm database.limit tables`. Either remove the `DependOn`, or add the referenced table to `limit tables`.

Vocabulary-loading errors

These appear when `automation.pre.populate vocabulary` is set but the engine cannot complete the load.

`Vocabulary directory empty / no zip found`

The directory pointed to by `vocabulary directory` is empty. Drop the Athena zip in there before the run.

ATHENA ZIPS WITH LICENSED CODE SETS (CPT-4, ETC.)

Some vocabularies — most commonly **CPT-4** — are not redistributable and Athena ships them as encrypted placeholders. The ETL Engine does **not** decrypt them automatically. The unpacking process is the same either way; only the last step differs depending on your CDM driver.

1. Download the vocabulary zip from Athena with the licensed code sets selected.
2. Unzip it to a working directory.
3. Run the unpacking scripts that ship inside the Athena zip (`cpt.bat` / `cpt.sh` or similar — Athena's own README in the zip has the canonical instructions). These scripts call out to the licensed code set's distribution API and replace the placeholders with real CSV rows.

CSV target (Free tier — no automation needed)

1. Copy the unpacked CSV files (`concept.csv` , `concept_relationship.csv` , `vocabulary.csv` , etc.) directly into `etc/etl-engine.d/cdm-out/` — the same directory the engine writes its CDM output to. The engine reads them in place; no automation step is involved.

RDBMS target (Paid tier — via automation)

1. **Delete the unpacking scripts** (`cpt.sh` , `cpt.bat` , any `*.jar` , `README` , or other non-CSV artefacts) before re-zipping. The engine's vocab loader reads every file in the zip as if it were a CSV, so a leftover shell script becomes a load-time crash.
2. **Repackage** the directory back into a zip — only `.csv` files inside. The engine reads from a zip, not from the unpacked directory.
3. Drop the new zip into the engine's `vocabulary directory` (`etc/etl-engine.d/vocab/`). The `populate vocabulary` automation will pick it up on the next run and import the rows into the CDM database.

Failure modes

- If you skip steps 2–3 entirely and let the engine consume the **original** Athena zip, the licensed concepts simply aren't loaded — no error, just rows missing from the `concept` table.
- For the RDBMS path: if you re-zip **with** `cpt.sh` / `cpt.bat` still present, the vocab loader crashes when it tries to parse one as a CSV.

`UsagiMap: ConceptID 'X' was not found in the vocabulary database`

The Usagi map maps a source value to a concept id that does not exist in the loaded vocabulary. Either re-export the Usagi map against a fresh vocabulary, or add the concept id manually. If the missing concept is in a licensed vocabulary like CPT-4, also check that you ran the unpacking process above before loading.

`UsagiMap: ConceptID 'X' does not have the domain id 'Y'`

The Usagi map points at a concept that is in a different domain than the rule's `AssertVocabDomain` requires. Re-curate the Usagi map or relax the domain assertion.

License errors

`License file invalid` / `License expired`

The license file the engine found is either malformed or has reached its expiry date. Contact your Rook IT representative to renew or replace it.

3.4.4 Performance tuning

Performance on a real ETL is multifactorial — disk speed, database speed, CPU count, CPU speed, and the engine's own code quality (a moving target — we ship optimisations regularly) all contribute to how long a run takes. There is no single setting that "makes the engine fast"; the discipline is **finding the current bottleneck and removing it**, then repeating against the next one.

Where to look

The bottleneck is usually one of these. The order below roughly matches "easiest to verify first":

CPU ON THE ENGINE HOST

Run `top` or `htop` while the engine is running. Two patterns to watch for:

- **All cores at 100%** — the engine is CPU-bound. Tuning won't help much; faster CPUs or splitting the workload across runs is the next move.
- **Only a few cores busy with workers idle** — the engine isn't fanning out. Check `worker threads` in `tool configuration`, and verify the source-row queue isn't starving (a slow source reader can leave workers waiting).

DESTINATION DATABASE

Watch the database while a run is in flight — `pg_stat_activity` on PostgreSQL, `SHOW PROCESSLIST` on MariaDB, the equivalent on SQL Server. Common patterns:

- **High lock waits / blocked queries** — the database can't keep up with parallel writers. Lower `worker threads`, or lower `max database connections` in `cdm database`, or both.
- **Slow individual inserts** — the destination has too many indexes or constraints active during the load. Drop non-essential indexes, load, rebuild — that is often a 10× speed-up on its own.
- **WAL/redo log pressure on PostgreSQL** — `synchronous_commit = off` for the duration of the load helps significantly. Revert afterwards.
- **Buffer-pool too small on MariaDB** — `innodb_buffer_pool_size` set to 60-70% of available RAM is the standard tuning.

DISK

Disk I/O matters in two places:

- **Source reads** when the source driver is `csv` and the files are on a slow disk or network mount.
- **Destination writes** when the database's data directory is on the same slow disk.

`iostat -x 1` shows whether disks are saturated. Move data to local NVMe if you can; failing that, the parallel-writer tuning above usually masks slow disks well enough.

CPU SPEED

If the run is CPU-bound and adding workers doesn't help (you've already saturated cores), single-thread performance becomes the ceiling. The engine processes per-row rule chains entirely on one worker thread per row — a faster CPU per core helps directly. If you can choose, machines optimised for single-thread performance beat machines with many slow cores.

CODE QUALITY

Engine performance is a moving target. We ship optimisations regularly, so a slow workload on one version may be much faster on the next without any configuration change. If a workload feels slower than it should, ask — we may already have an improvement in flight, or the workload itself may be exposing a hot spot worth fixing.

Configuration knobs

These are the engine-side levers, in roughly the order to reach for them:

`worker threads`

The single most impactful knob. Set in `tool configuration`.

```
tool configuration {
  worker threads: 10
}
```

Each worker pulls source rows and runs the rule chain independently. More workers = more parallelism, until CPU, the destination database, or `max database connections` becomes the limit.

`max database connections`

```
cdm database {
  max database connections: 4
}
```

Caps how many connections the engine opens to the destination. Match it to what your database can comfortably handle without piling up lock waits.

`vocabulary cache size`

```
cdm database {
  vocabulary cache size: 5000
}
```

Memory for the OMOP-vocabulary lookup cache. Helps when many rule chains hit the OMOP vocabulary (`AssertVocabDomain`, `VocabMap`) and the same concept ids are looked up repeatedly. Memory grows linearly with the setting.

`use dedup cache`

```
tool configuration {
  use dedup cache: true
}
```

Caches deduplication results across rows. Helpful when many source rows resolve to the same `Normalize` key. **Disabled by default** because the memory cost isn't worth it on workloads with low duplication.

Splitting the load

For very large ETLs, splitting the mapping across multiple Rabbit-in-a-Hat files and several engine runs is often the cleanest answer: each run is shorter, individually re-runnable, and easier to profile. The `StoreIndexMap` and `StoreSequence` rules persist state between runs — see Folder Layout → Splitting an ETL.

When to ask for help

If you've worked through the checks above and the bottleneck still isn't obvious — or it is obvious and the answer needs an engine-side fix — get in touch. Performance work on real customer data is one of the things a Paid license buys you.

3.4.5 TraceWhen

`TraceWhen` is a table rule that enables **per-record verbose logging** when a source field matches a list of values. It is the targeted-debugging tool of choice when one specific row, or a small set of rows, behaves differently from the rest.

Usage

Add `TraceWhen` to a CDM table's Comments field with `logic:`:

```
logic: TraceWhen(:patient_id, ["abc-123", "def-456"])
```

The first argument is the source field to match against; the second is a list of values that activate tracing. When any source row has a matching value, every rule that runs for that row writes a detailed trace line to the per-table log.

What gets logged

Each rule's name and inputs/outputs are logged for the matched rows. Rule names are demangled, so the log shows the DSL name (e.g. `UsagiMap`) rather than the C++ symbol (`_ZN8UsagiMap...`).

Performance impact

Tracing has a per-row cost only for rows that match. Non-matching rows go through the normal fast path. It is safe to leave a `TraceWhen` rule in place during a long run as long as the value list is small.

Limitations

Tracing is automatically disabled inside recursive `additionalQueue` mapping calls, to prevent re-entrant logging from corrupting the trace output. This is intentional — if you need to trace inside one of those calls, restructure the chain so the tracing happens at the outer scope.

4. Examples

4.1 Examples

Worked examples — end-to-end walkthroughs that take a real dataset and follow it from source files all the way to a populated OMOP CDM. The examples are deliberately the same shape: introduce the dataset, map one table at a time, accumulate techniques, then end with a "useful patterns" page that shows the corners of the language.

Examples are open-ended — this section is meant to grow over time as new mapping patterns prove worth writing up.

4.1.1 Currently in this section

Patterns

Self-contained recipes for the engine's flow-control features — multi-result mappings, conditionals (`If` / `Else` / `Switch`), fallback chains (`||`), and parallel collection (`++`). One feature per page; each example is lifted from a real test in the repo's test suite.

Synthmas

A synthetic Massachusetts patient dataset (CSV files generated by Synthea). The Synthmas walkthrough is inherited from the v1.3 PDF and was the original "let's actually explain how to use the engine" content.

It walks through:

- Overview — the source schema, the target CDM, and the mapping plan.
- The location table — table mapping, multiple sources, normalisation, type mismatches, padding, and hard-coded fields.
- The person table — vocabulary maps with default-and-assert, date deconstruction, ID generation, and cross-table lookups.
- Useful patterns — vocabulary lookups, Usagi maps, conditional chaining.

4.1.2 Adding new examples

The Synthmas walkthrough is a template, not a ceiling. New examples should follow the same structure — an `index.md` introducing the dataset and the mapping plan, one page per CDM table that benefits from a walkthrough, and a closing page on patterns that emerged.

When in doubt, mirror the Synthmas overview.

4.2 Synthmas — overview

Synthea is an open-source synthetic patient generator from MITRE. **Synthmas** is the dataset that comes out when you run Synthea against the population of Massachusetts — synthetic patients with synthetic encounters, conditions, observations, medications, devices, and so on. No real-patient data is involved, which makes it the ideal demonstration corpus for the engine.

The Synthmas extract used in this walkthrough ships as a directory of CSV files — one per source table — which is exactly what the engine's CSV driver consumes.

Get the bundle

The **Synthmas CSV + Rabbit-in-a-Hat bundle** is a free download from rook-it.com/downloads — a single archive with the source CSVs, the single-RiaH file used in this walkthrough, and a working `etl.conf`. No login required for the latest version. With the Free tier and no licence file, you can have a populated CDM out of the box and follow this guide step-by-step against your own running engine.

Already have an engine install? Grab the overlay tarball `synthmas-single-RiaH-1.0.21.tar.gz` instead — the RiaH, Usagi maps, and `etl.conf` only (no CSVs). Overlay it onto the engine artefact with `tar xzf synthmas-single-RiaH-1.0.21.tar.gz --strip-components=1`, then fetch the source CSVs with `bin/setup-synthmas.sh`.

4.2.1 Source schema

The relevant source tables for this walkthrough:

Source file	Description
`patients.csv`	One row per patient — id, dob, gender, race, address.
`organizations.csv`	Provider organisations — also addresses.
`providers.csv`	Individual providers — addresses again.
`encounters.csv`	Visits between patients and providers.
`conditions.csv`	Conditions diagnosed during encounters.
`procedures.csv`	Procedures performed during encounters.
`observations.csv`	Observations recorded during encounters, with units.
`medications.csv`	Drug exposures.
`devices.csv`	Devices issued to patients, with UDI codes.
`payers.csv`	Insurance organisations — one row per payer, with a `name` column used as the source code in the payers Usagi map.
`payer_transitions.csv`	One row per `(patient, payer, year-range)` — the input that drives `payer_plan_period`.

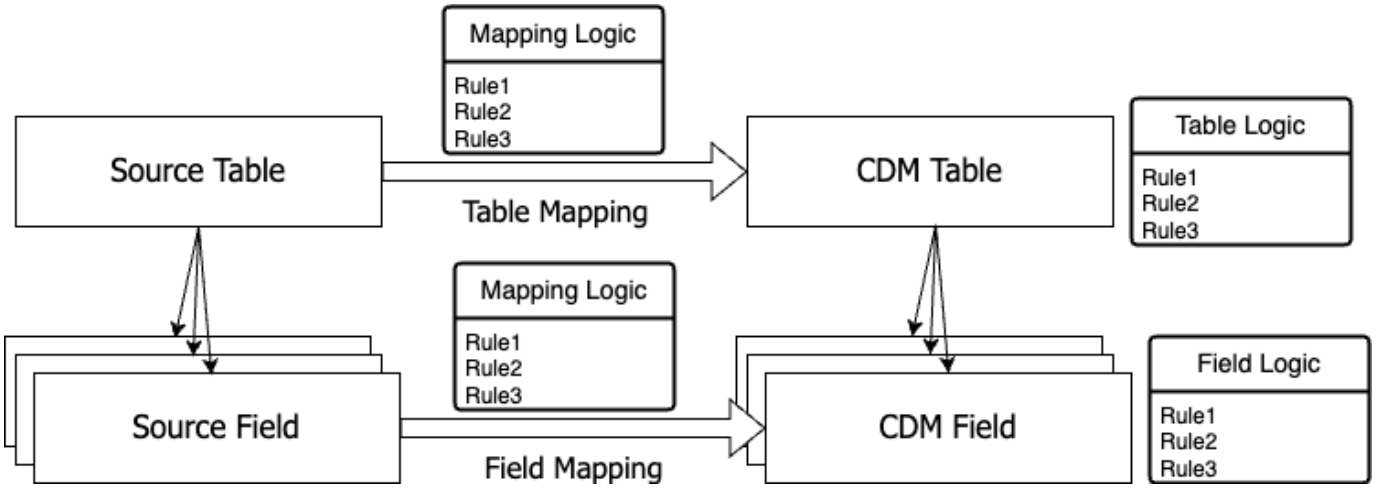
4.2.2 Target CDM

The walkthrough populates a representative slice of the OMOP CDM — twelve real CDM tables plus three that are derived rather than read directly from any single source:

CDM table	Notes
`location`	Addresses extracted from several source tables. Forces normalisation.
`person`	One row per patient.
`care_site`	Provider sites.
`provider`	Individual providers.
`visit_occurrence`	Encounters → visits.
`visit_detail`	Sub-visits within an encounter.
`condition_occurrence`	Diagnoses with SNOMED-coded concepts.
`procedure_occurrence`	Procedures with SNOMED codes.
`observation`	Observations with mode-dispatch into `value_as_number` / `value_as_string`.
`measurement`	Numeric measurements with LOINC vocabulary lookups.
`drug_exposure`	Medications, with a date-arithmetic pattern for the end date and UDI parsing.
`device_exposure`	Devices with a fixed-interval end date and UDI parsing.
`death`	**Derived** — built from a `LeftJoin` of patients with conditions.
`observation_period`	**Derived** — observation windows tied back to person.
`payer_plan_period`	**Derived** — driven by `payer_transitions.csv`, `LeftJoin`-ed against `payers.csv` for the payer name; payer concept id via Usagi map.

The target database in this walkthrough is **PostgreSQL**, with the CDM tables in the `omop_cdm` schema.

4.2.3 Mapping plan



A mapping plan starts in Rabbit-in-a-Hat:

1. Draw the source-to-CDM table arrows for the tables you intend to populate. The simple cases (one source → one CDM) need no rule at all.
2. Draw the field-to-field arrows where source and CDM types match.
3. Add a `logic:` rule on a CDM field where a transformation is needed (type cast, padding, vocabulary lookup, …).
4. Add a `logic:` rule on a CDM table where a table-level operation is needed (`Normalize`, `DependOn`, `Join`, `LeftJoin`).

4.2.4 Walkthrough order

The pages below build up techniques by introducing one or two new patterns each:

- The location table — `Normalize`, type mismatches, padding, defensive `Truncate`, hard-coded fields. The warm-up.
- The person table — vocabulary maps with default
- assert, date deconstruction, ID generation, cross-table lookups.
- Clinical events — `visit_occurrence`, `condition_occurrence`, `procedure_occurrence`. `Selector` based on a class field; pre-map + `VocabMap` for codes that aren't in the standard vocabulary; the universal `type_concept_id` pattern.
- Values and dates — `observation`, `measurement`, `drug_exposure`, `device_exposure`. Mode-dispatch on `:type`, date arithmetic for end dates, UDI regex parsing, null fallback with `Selector`.
- Derived tables — `death`, `observation_period`, `payer_plan_period`. `LeftJoin` to derive a CDM table from two source tables (twice — `death` joins on a shared date, `payer_plan_period` joins a fact table against a lookup); cross-CDM foreign keys with `GetCDMTableId(@person) + AssertNot(NULL)`.
- Useful patterns — vocabulary lookups, Usagi maps, conditional chaining: the toolkit you'll reach for over and over.
- FAQ — questions that come up the first time you watch `run.log` (record counter past 150,000, throughput swings, queue labels, unmapped concept ids).

4.2.5 How to follow along

If you want to run the example yourself:

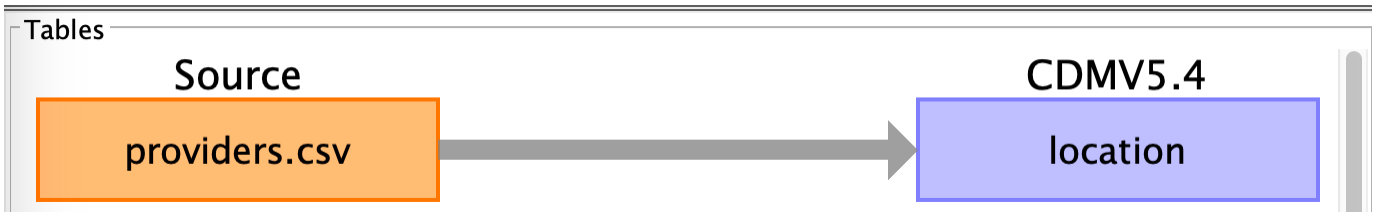
1. Get the Synthmas bundle (or generate your own with `synthea -p 50 Massachusetts`).
2. Drop the CSVs into `etc/etl-engine.d/db/` of the engine artefact — see Folder Layout.
3. The bundled Rabbit-in-a-Hat file goes alongside `etl.conf`.
4. Run `bin/run-etl-engine.sh` and watch `logs/<table>.log` for the results.

4.3 Synthmas — the `location` table

We start with `location` because it forces every interesting case the engine has to handle: multiple source tables feeding one CDM table, type mismatches, normalisation, and hard-coded fields.

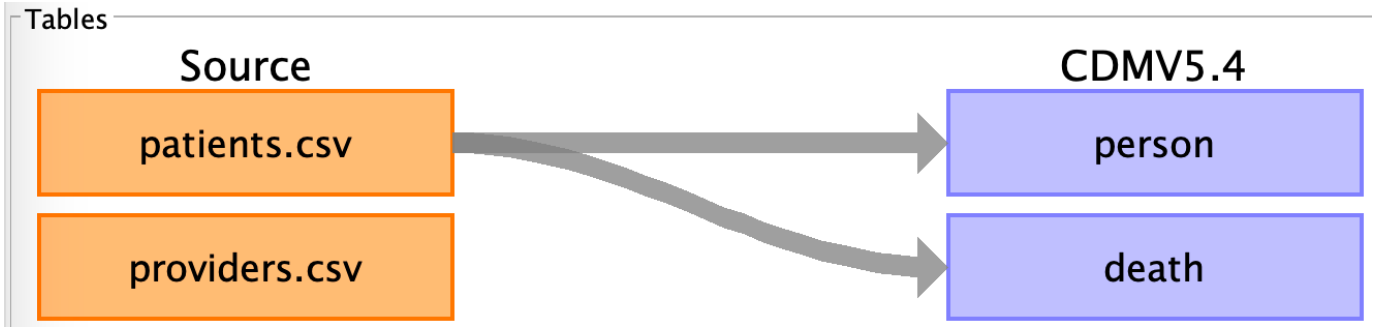
4.3.1 Step 1 — the simplest possible mapping

The simplest mapping is one source table to one CDM table, drawn as a single arrow in Rabbit-in-a-Hat:



This makes the engine copy `providers.csv` records into the `location` table, on the condition that all required CDM fields have a mapping — more on that in a moment.

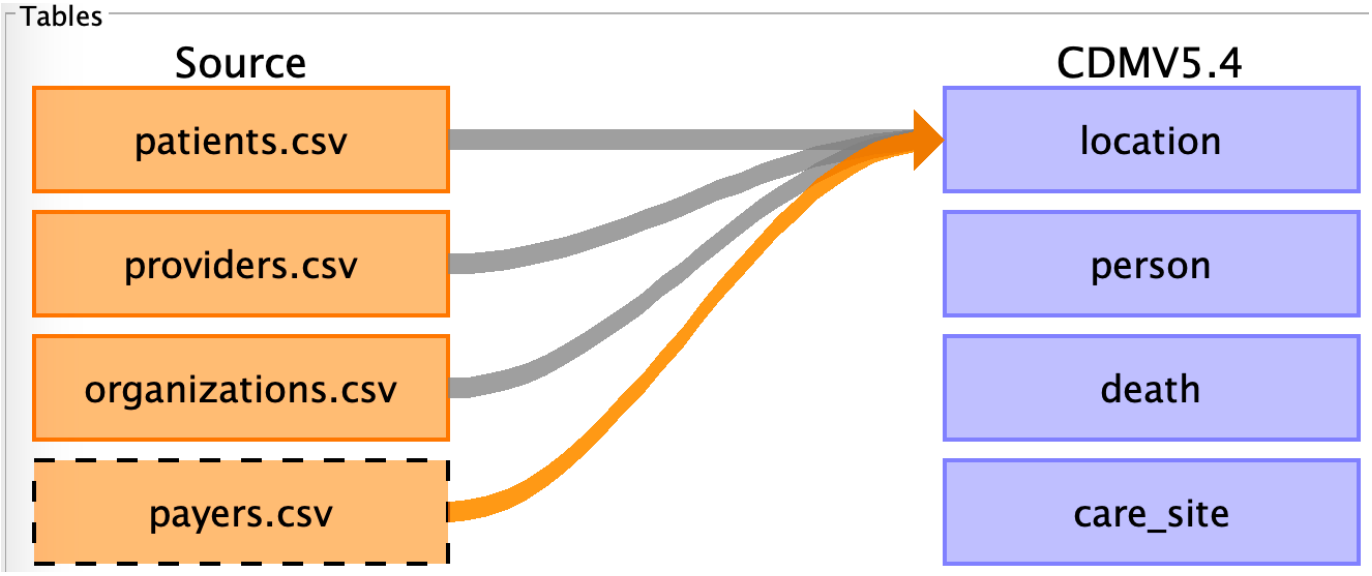
A close cousin is the *one source, multiple CDM tables* shape — also visual, also rule-free:



The mapping in the figure above sends `patients.csv` to both `person` and `death`. Even though most patients have not died, the mapping must still exist so that those who have are mapped — gating "alive vs dead" happens in field rules, not at the table level.

4.3.2 Step 2 — multiple sources to one CDM table

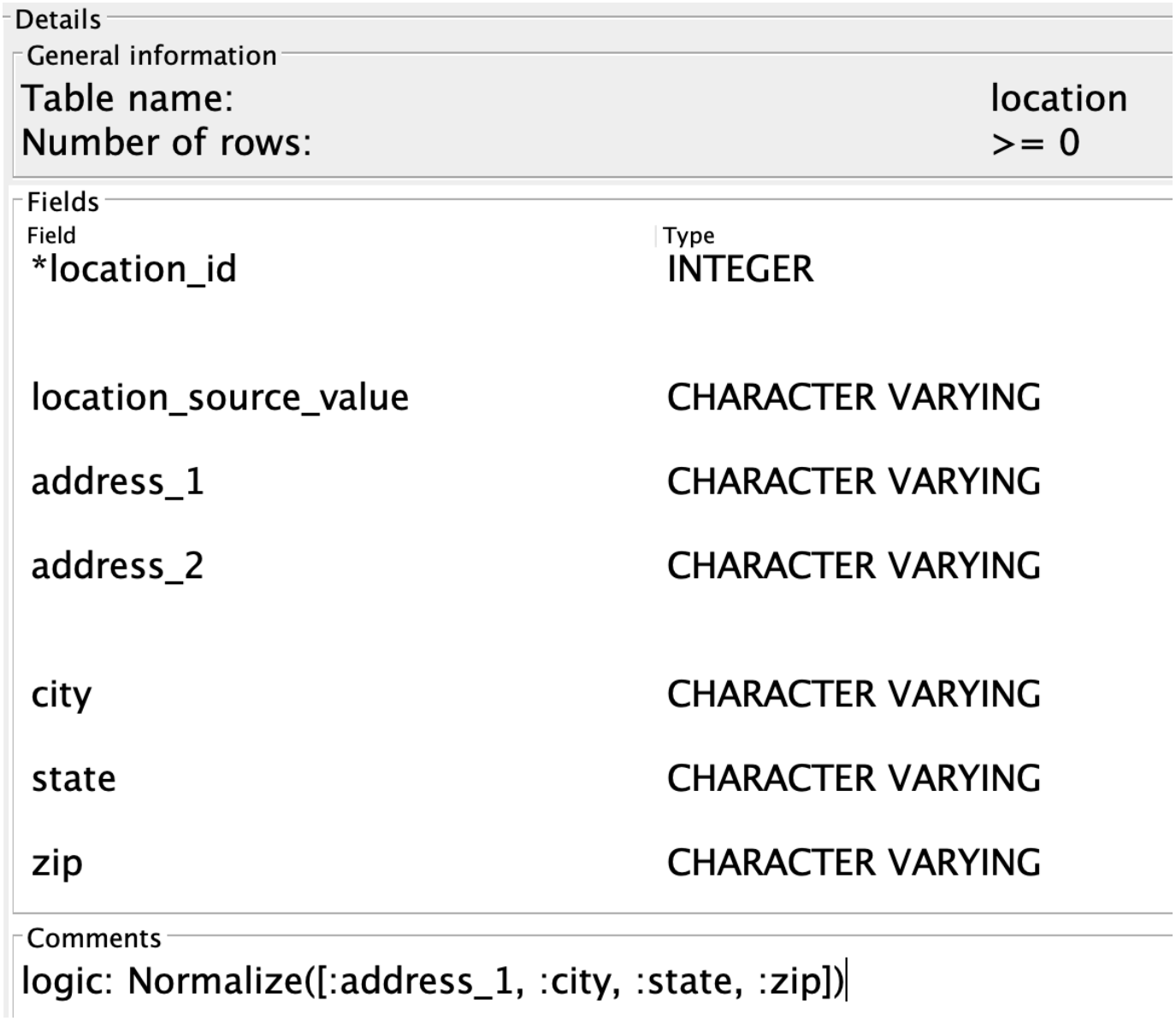
The opposite shape — several source tables into one CDM table — is where the engine starts to need help. Synthmas has location-shaped data scattered across several source tables:



Without any further configuration, the engine would simply concatenate records from all four source tables into `location`. That produces duplicates, because providers and organisations frequently share addresses, and the same address appears repeatedly across rows of `providers.csv`.

The fix is the `Normalize` table rule. `Normalize` builds a unique-key index on the `location` table; new records that match an existing key on those fields are silently dropped.

Add it to the `Comments` field of the CDM `location` table, prefixed with `logic:` (since that field is normally for free-form comments, the prefix tells the engine that what follows is a rule):




```
logic: Normalize([:address_1, :city, :state, :zip])
```

`Normalize` is useful even when only one source table feeds `location` — providers with the same office share an address.

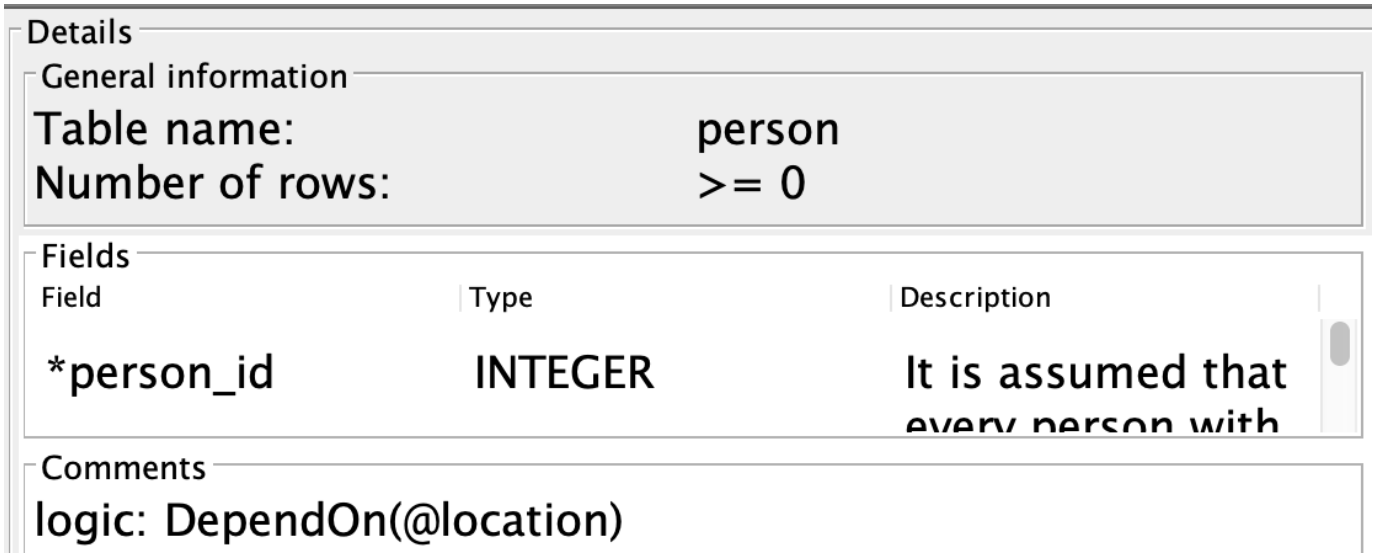
> **Where rules can live on a CDM table**
>
> The `Comments` field of any CDM table accepts table-level rules (with the `logic:` prefix) — see Appendix B → Table rules. The arrow from a source table to a CDM table also accepts rules — see Appendix A → Filter rules.

4.3.3 Step 3 — joins between source tables

Sometimes the records from several source tables need to be combined *before* they enter the CDM. That's what `Join` and `LeftJoin` are for. They work like SQL joins — the syntax is different but the semantics are the same. They can be nested by giving the joined result a name and referencing it from a later `Join` / `LeftJoin`.

4.3.4 Step 4 — declared dependencies

When the engine cannot infer a load order from the Rabbit-in-a-Hat arrows alone — typically because a CDM table has a foreign key into another CDM table that is not visible to the engine — use `DependOn`:

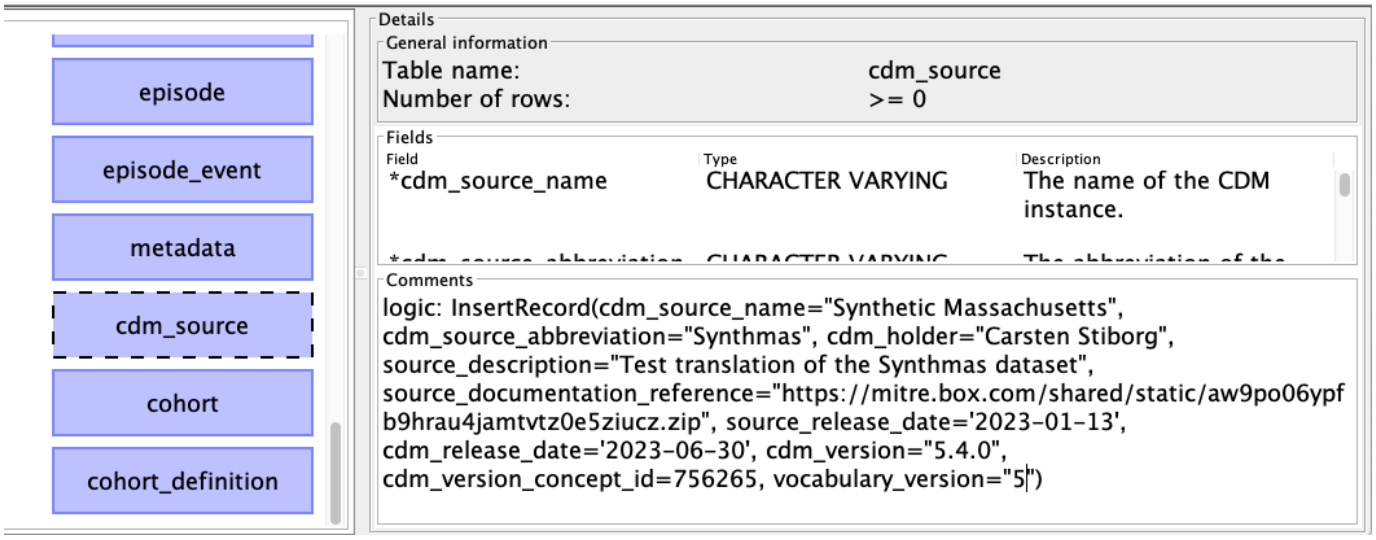



```
logic: DependOn(@location)
```

> **Cyclic dependencies**
>
> `DependOn` lets you build a cycle between CDM tables. The engine does not break cycles — review your `DependOn` rules carefully.

4.3.5 Step 5 — static records

Some CDM tables hold a fixed set of rows that are not derived from the source data — `cdm_source` is the canonical case, with a single row describing the data source itself. Use `InsertRecord`:



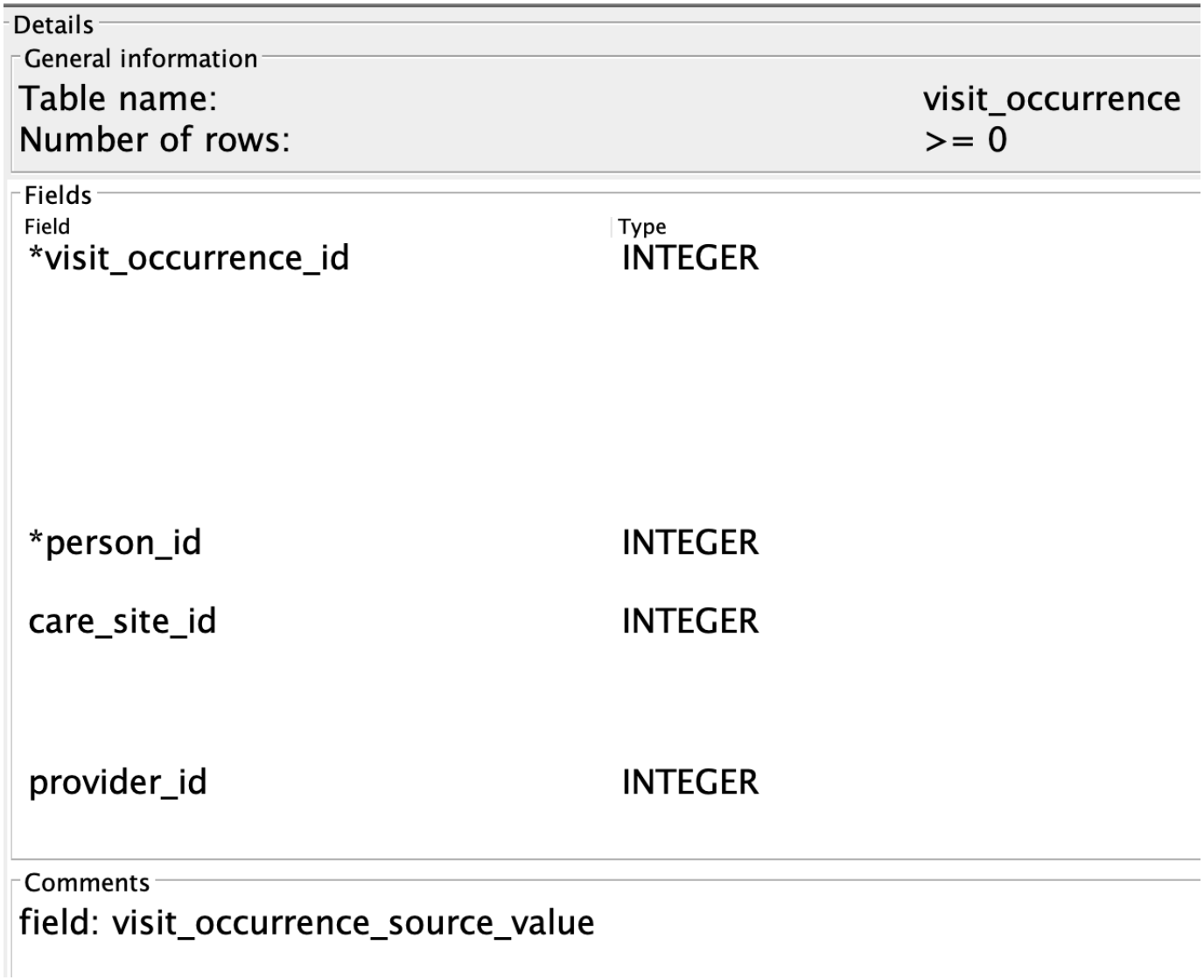

```
logic: InsertRecord(
  cdm_source_name="Synthetic Massachusetts",
  cdm_source_abbreviation="Synthmas",
  cdm_holder="Rook IT ApS",
  source_description="Synthmas dataset translation",
```

```
    cdm_release_date='2023-06-30',
    cdm_version="5.4")
```

Every non-nullable field on the table must be supplied as a `field=value` keyword parameter.

4.3.6 Step 6 — a virtual field

Virtual fields are CDM-table fields that exist only inside the engine. They never get written to the destination database, but rule chains can read and write them like ordinary fields. Declare one with `field:` instead of `logic:` in the CDM table's `Comments` field:



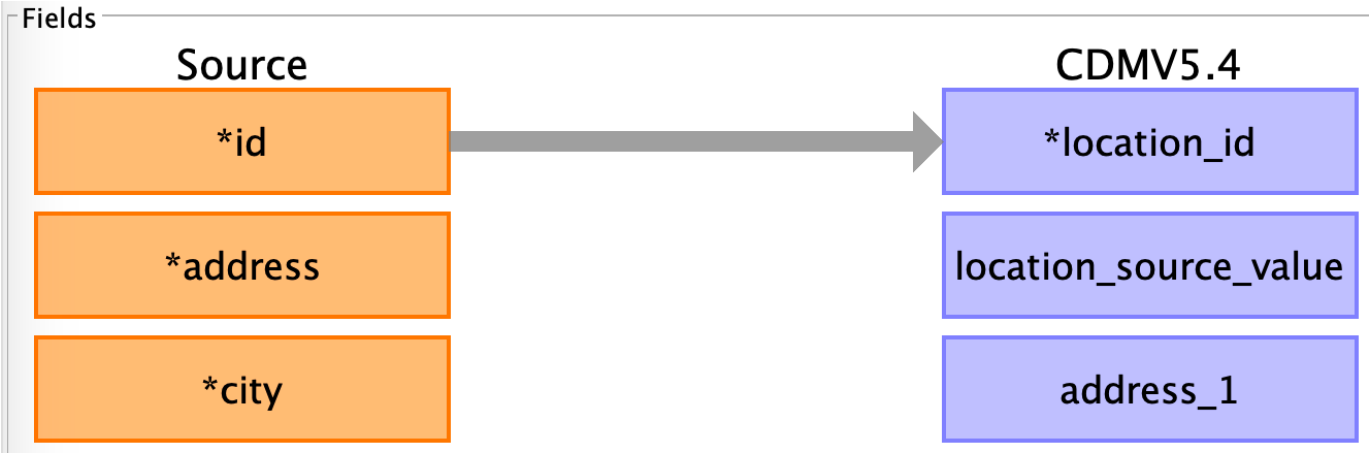

```
field: visit_occurrence_source_value
```

The text after `field:` becomes the field name. Above we declare a `visit_occurrence_source_value` field on `visit_occurrence` — useful because CDM 5.4's `visit_occurrence` is the only table without a built-in `*_source_value` field for its primary key.

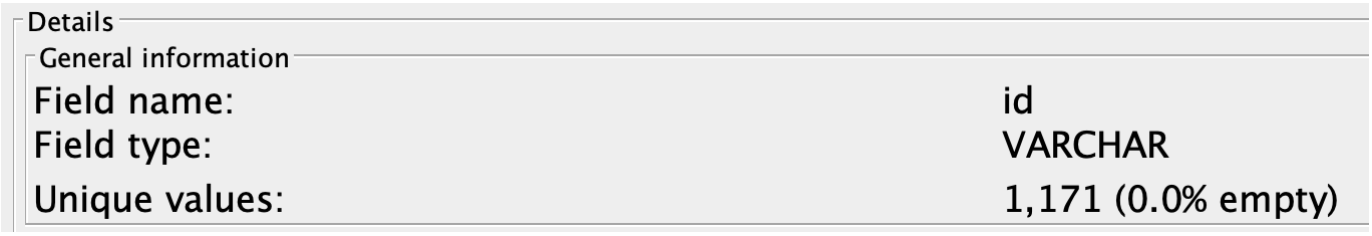
For more on virtual fields, see DSL Guide → Virtual Fields.

4.3.7 Step 7 — minimal field mapping

Now that the table-level scaffolding is in place, we wire up the fields. The minimum requirement is that every non-nullable CDM field has *some* mapping — anything else (the optional fields) can be omitted.



In the figure above, `id` (source) is mapped to `location_id` (CDM). **This won't actually work** — and looking at the source schema tells us why:

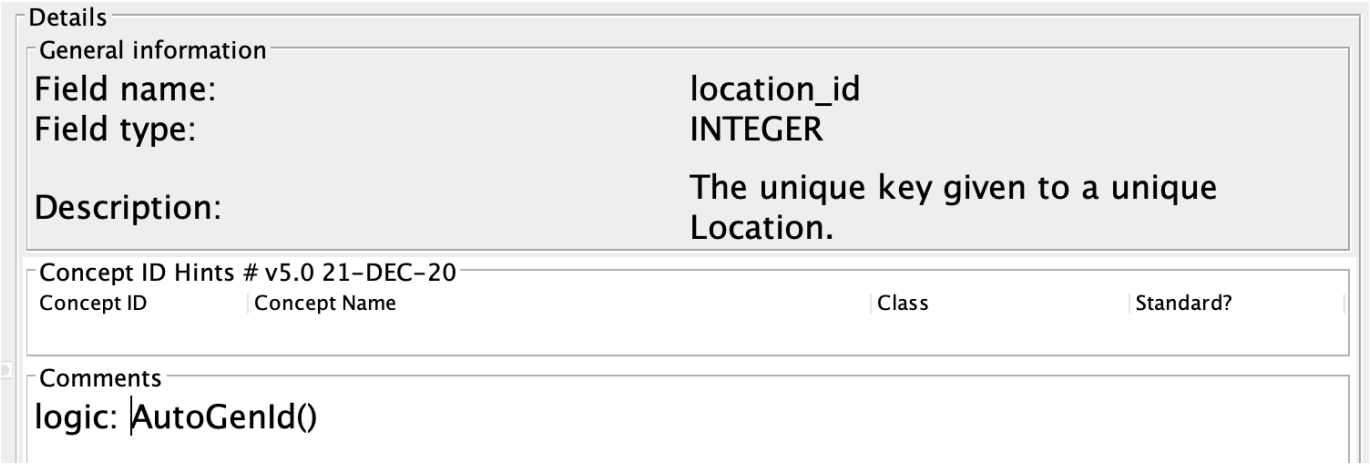


The source `id` is a UUID *string*, while CDM `location_id` is an integer. We have two options:

1. Map the UUID to a *different* CDM field — `location_source_value` is a string and exists for exactly this purpose. Then generate a fresh integer `location_id`.
2. Refuse to map the row.

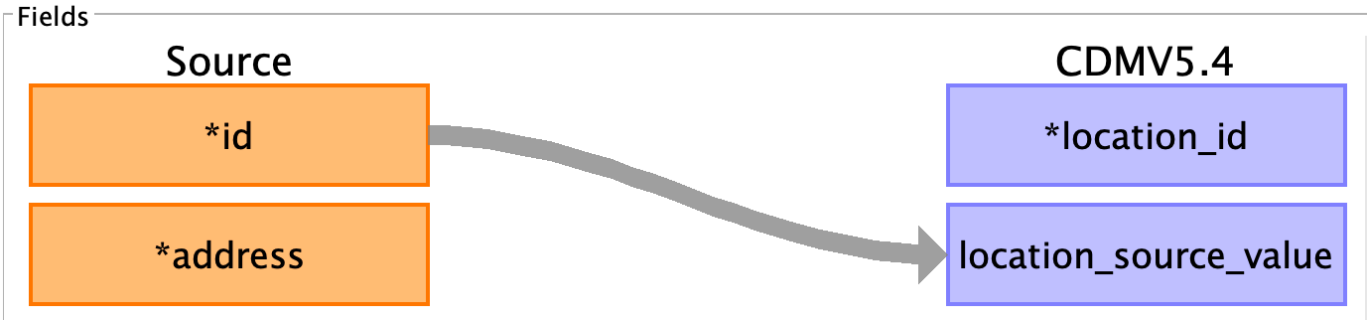
Option 1 is the standard pattern. It also generalises to the case where multiple source tables have IDs that aren't unique across them — prefix or postfix the source ID with a table identifier before storing it in `location_source_value`, then generate an integer `location_id` independently.

The integer comes from `AutoGenId` — added to the `Comments` field of the CDM `location_id` field with the `logic:` prefix:




```
logic: AutoGenId()
```

`AutoGenId` issues a fresh, monotonically increasing integer for every row that reaches it. Combined with the source UUID stored in `location_source_value`, every row of `location` now has a unique CDM key plus the original source identifier preserved for traceability.



4.3.8 Step 8 — type-safe field mapping

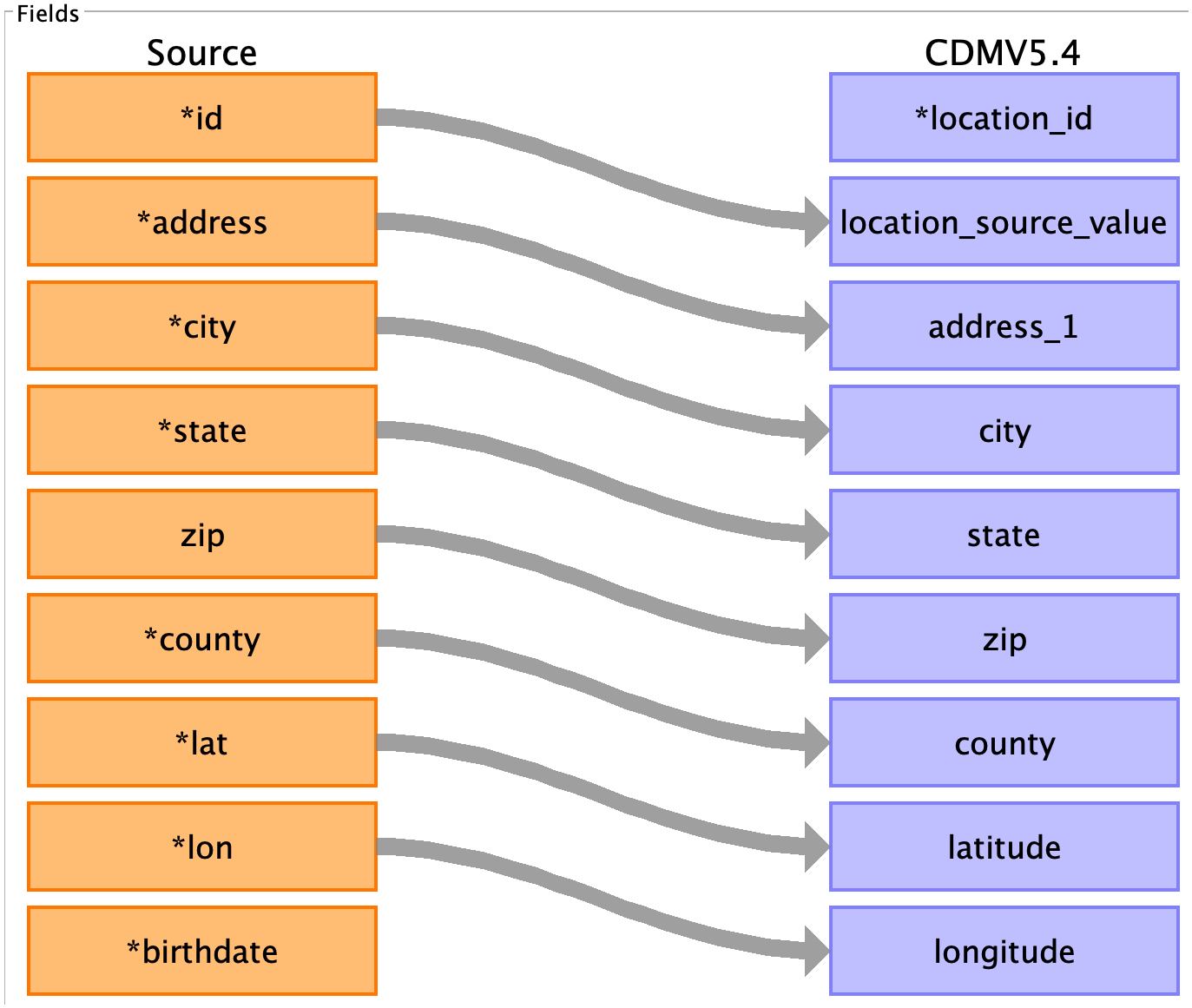
Most field-to-field mappings will work out of the box if the source and CDM types match. For Synthmas's `location` table, two mappings need work:

- **`zip`** → **`zip`** — source is integer, CDM is string. We also want to pad short zip codes with leading zeros and cap the length.
- **`state`** → **`state`** — source is the full state name (`"Massachusetts"`), CDM is a 2-character code.

Zip: cast and pad

The CDM `zip` field is a string of up to 9 characters. US zip codes are at most 5 digits without routing info, but a leading zero might be missing if it was stored as an integer. So: cast to string, then pad with leading zeroes to length 5 if shorter.

This goes on the field-to-field arrow, in a logic block:




```
AsString()
PrePad(5, "0")
```

Inside a `logic:` block on a field-to-field arrow, no `logic:` prefix is needed on each line — the block itself is the prefix.

Zip: a defensive normalisation on the CDM field

The CDM `zip` field is a string. The Synthmas source has zip codes in mixed formats — some are bare 5-digit codes, some are the routed `12345-6789` form, and the patient-table version is stored as an integer that loses any leading zero. We're not going to fight the data: just truncate to the canonical 5-digit form on the way in.

A `Truncate` rule directly on the CDM `zip` field applies to *every* incoming value, regardless of which source table it came from:


```
logic: Truncate(5)
```

This is also the difference between a rule on a *map* and a rule on a *field*. A rule on a map applies to one specific source-to-CDM transition; a rule on a field applies to every value that ends up in the field, no matter where it came from.

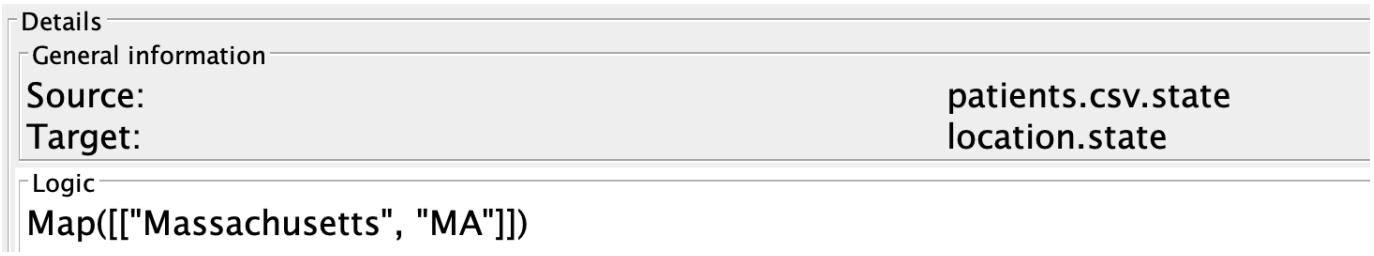
State: vocabulary map

Mapping `"Massachusetts"` → `"MA"` is a static lookup. Multiple source tables feed `state`, and most use the two-letter form already. So the mapping goes on the source-to-CDM arrow that needs the conversion, not on the field itself:


```
Map([
  ["Massachusetts", "MA"]
])
```

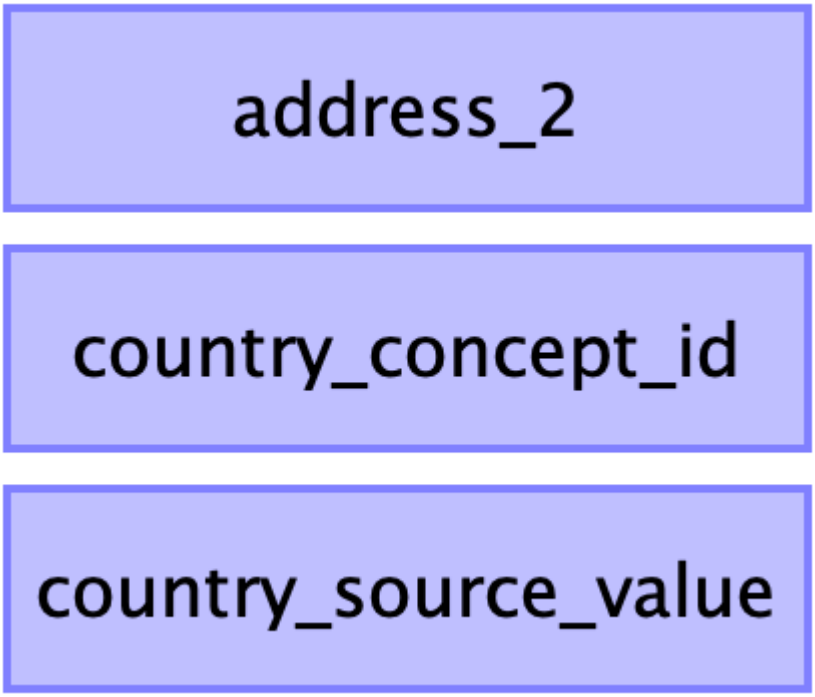
4.3.9 Step 9 — hard-coded fields

After mapping the source fields, three CDM fields remain unmapped:



`country_concept_id`, `country_source_value`, and `latitude` / `longitude` are not present in the Synthmas source. None are required, so leaving them out is fine — but we know every Synthmas address is in the US, so we may as well hard-code `country_concept_id`.

A hard-coded value goes on the *target field* itself. There is no field-to-field arrow into `country_concept_id`; the rule is the mapping:




```
logic: SetValue(42046186)
```

`42046186` is the OMOP standard concept id for the United States, found via Athena. Pair it with a matching `country_source_value` so the original string is preserved:

```
logic: SetValue("United States of America")
```

4.3.10 Recap

The `location` table now has:

- a `Normalize` rule on the table for de-duplication,
- field-to-field arrows where source and CDM share a type,
- `AutoGenId` on `location_id` for fresh integer keys,
- the source UUID preserved in `location_source_value`,
- `AsString` + `PrePad` on the zip-to-zip arrow,
- a defensive `Truncate(5)` on the CDM `zip` field,
- `Map(...)` on the state-to-state arrow,
- `SetValue(42046186)` on `country_concept_id` and a matching `country_source_value`.

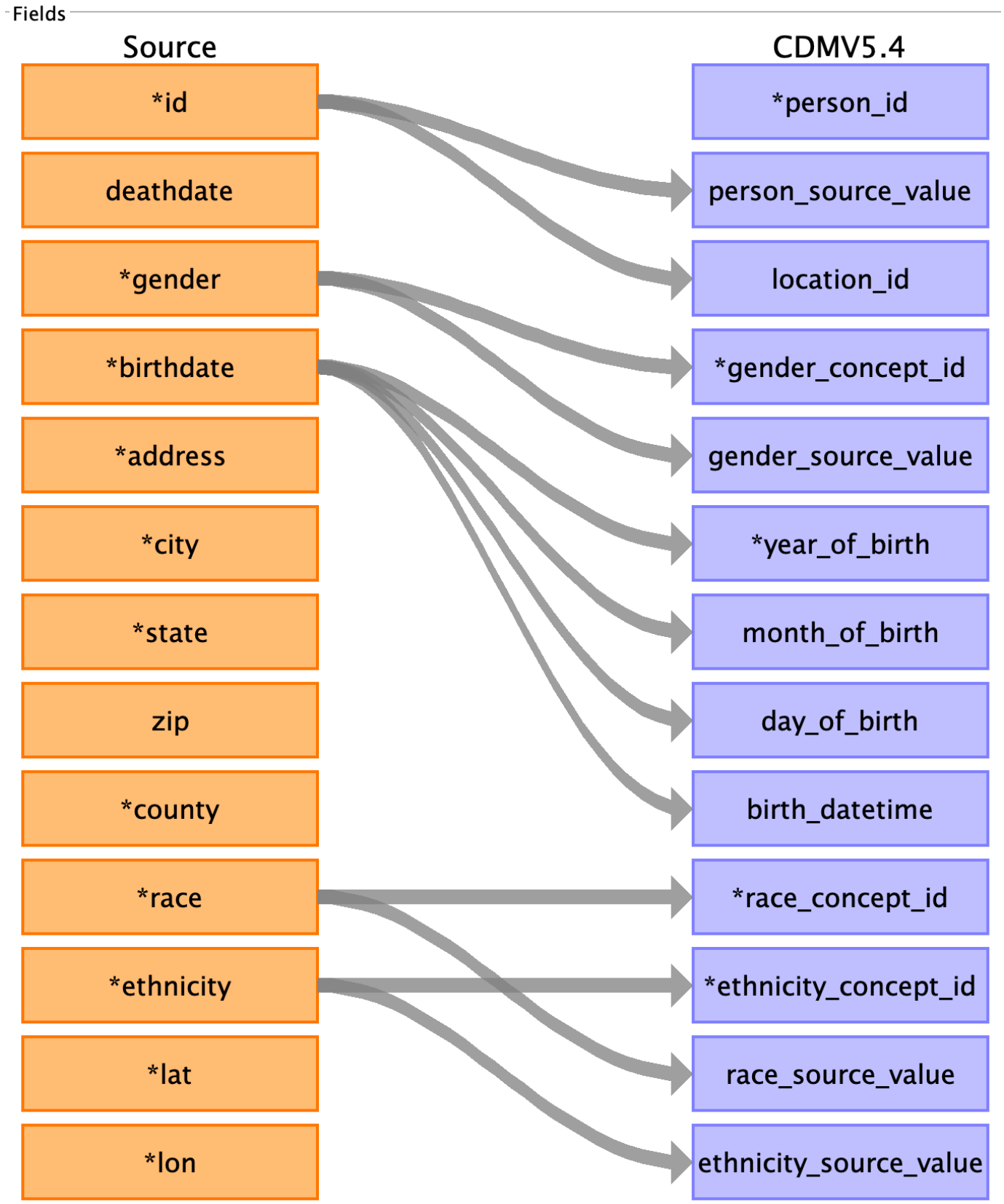
That's the location table done. The person table is next, and adds vocabulary mapping, date deconstruction, and a cross-table lookup on top.

4.4 Synthmas — the `person` table

The `person` table is the most central table in the OMOP CDM — practically every clinical event references it via `person_id`. We'll map Synthmas's `patients.csv` into it, and along the way introduce:

- vocabulary lookups via `Map` with a default plus an `AssertNot` on the default,
- date deconstruction with `YearFromDate` / `MonthFromDate` / `DayFromDate`,
- ID generation with `AutoGenId` (recap from `location`),
- and a cross-table lookup with `GetCDMTableId`.

4.4.1 Field overview



The figure shows the field-to-field arrows for the `person` table. A few fields are not in the figure for space reasons (the various `*_source_concept_id` columns and `provider_id` / `care_site_id`); none are mapped in this walkthrough — Synthmas's gender, race, and ethnicity values are all strings, not concept ids, so the `*_source_concept_id` slots stay null.

4.4.2 Vocabulary lookups: `Map` + default + `AssertNot`

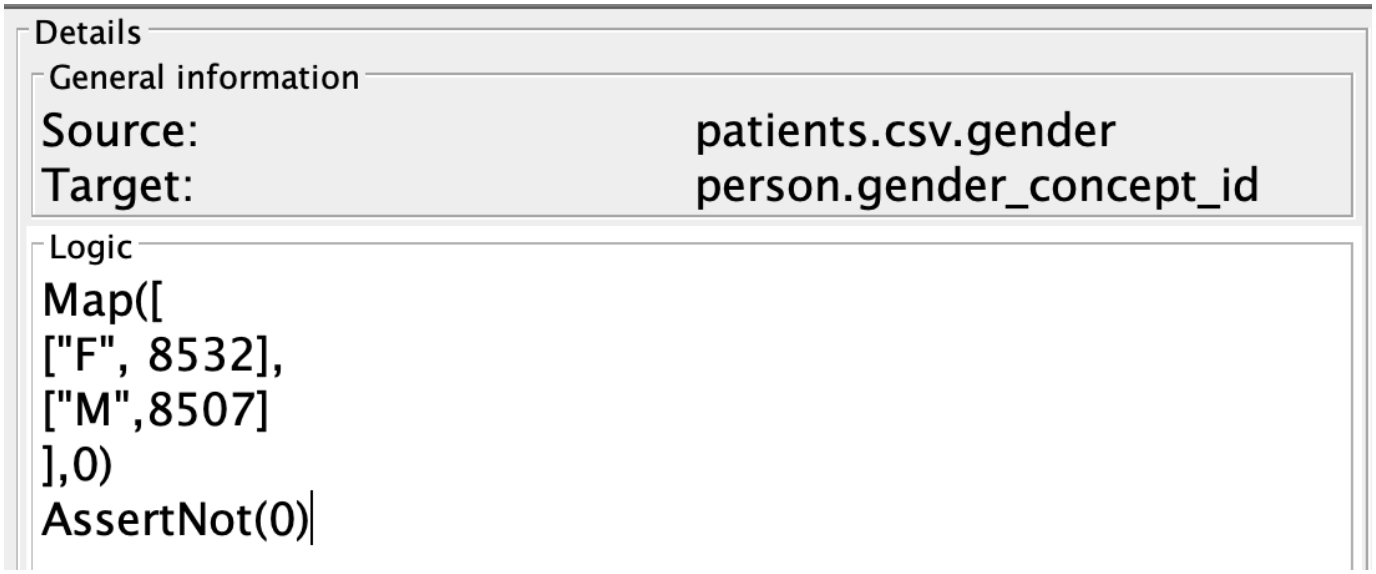
Three fields need vocabulary lookups:

- **gender** — `M` / `F` → `gender_concept_id`.
- **race** — `white` / `black` / `asian` / `native` → `race_concept_id`.
- **ethnicity** — `nonhispanic` / `hispanic` → `ethnicity_concept_id`.

The pattern is the same for all three. Use a `Map` with a default value, then chain an `AssertNot` that fails the row when the default has been used. This way:

- Values you expected get mapped to the right concept id.
- Values you didn't expect end up with the default, fail the assert, and get logged — so you know to add them to the `Map`.

Gender:




```
Map([
  ["M", 8507],
  ["F", 8532]
], 0)
AssertNot(0)
```

Race:

Details

General information

Source: patients.csv.race
Target: person.race_concept_id

Logic

```
Map([
["White", 8527],
["Black", 8516],
["Asian", 8515],
["Native", 8657]],
0)
AssertNot(0)
```

```
Map([
  ["white", 8527],
  ["black", 8516],
  ["asian", 8515],
  ["native", 8657]
], 0)
AssertNot(0)
```

Ethnicity:

Details

General information

Source: patients.csv.ethnicity
Target: person.ethnicity_concept_id

Logic

```
Map([
["nonhispanic", 38003564],
["hispanic", 38003563]
], 0)
AssertNot(0)
```

```
Map([
  ["nonhispanic", 38003564],
  ["hispanic", 38003563]
], 0)
AssertNot(0)
```

For each of the three sources, there is also a parallel arrow to the matching `*_source_value` field with no rule — that preserves the original string so it survives into the CDM.

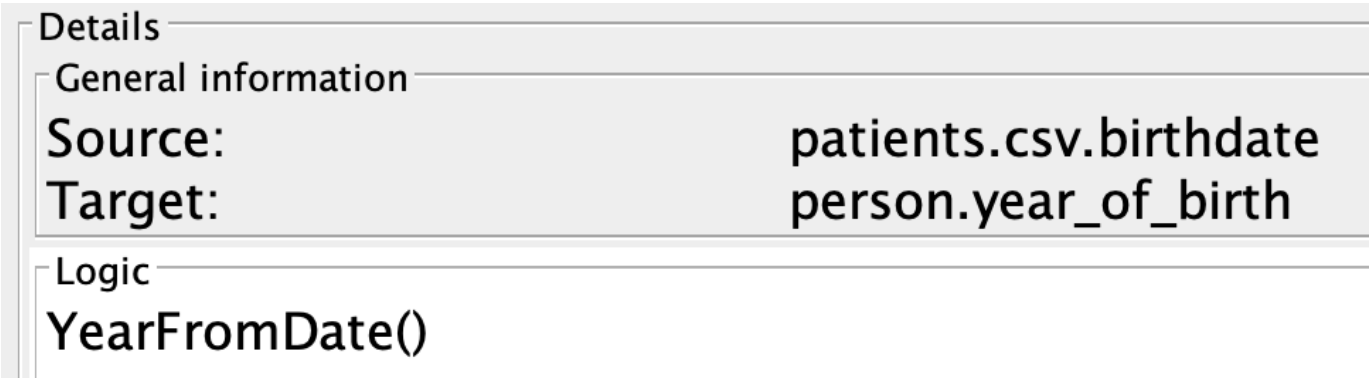
> ⚠ **Why default + assert, not just `Map(...)`**
>
> Without a default, `Map` does **not** raise an error on an unmapped input — it **passes the original value through unchanged** and sets `successfulMapping = false`.
>
> Inside a `||` chain that's the cue for the next conditional rule to fire. Outside a `||` chain it's a footgun: an unmapped `STRING` like `"unknown"` flows on into a CDM `INT` field, which either errors at the load boundary or — depending on the driver's strictness — inserts garbage. Big KA-BOOM.
>
> Default + `AssertNot(default)` short-circuits both failure modes:
>
> - The default sentinel (here, `0`) is a value of the correct *type*, so type checks pass.
> - `AssertNot` rejects exactly that sentinel and logs the row, so unmapped values land in the table log instead of the CDM table.
>
> Use this pattern any time `Map` is the last rule in a chain.

4.4.3 Date deconstruction

Synthmas has `birthdate` as a date, but the CDM `person` table wants it split into `year_of_birth` (mandatory), `month_of_birth`, `day_of_birth`, and `birth_datetime` (the original date).

`birth_datetime` is a straight field-to-field arrow with no rule — both sides are dates.

The other three need `YearFromDate`, `MonthFromDate`, and `DayFromDate`:




```
YearFromDate()
```

Each rule sits on its own field-to-field arrow from `birthdate` to the matching CDM field. They take no parameters.

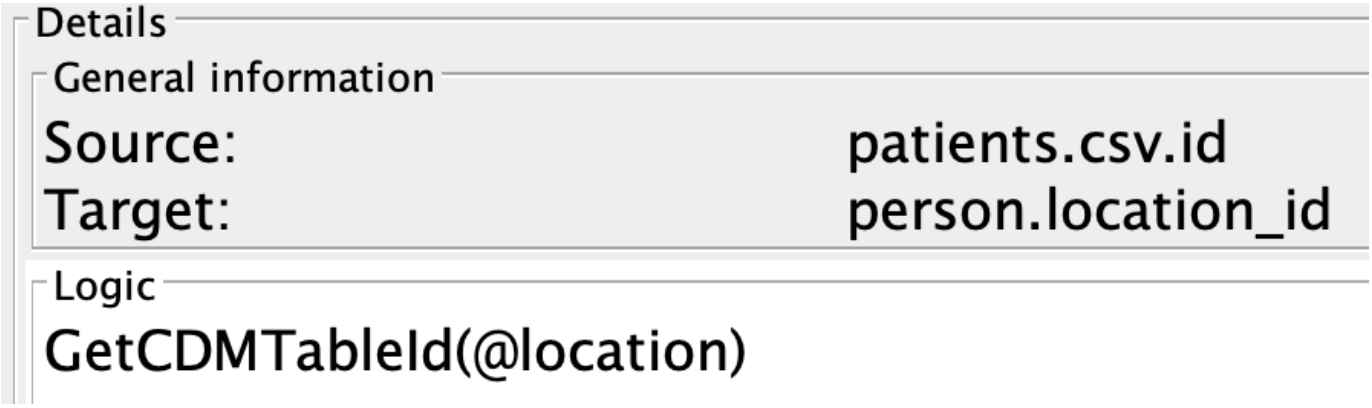
4.4.4 `person_id` and `location_id`

`person_id` follows the same pattern as `location_id`:

- Map the UUID `id` from `patients.csv` to `person_source_value` (string-to-string, no rule).
- Add an `AutoGenId` on the CDM `person_id` field.

`location_id` is more interesting. The CDM `person.location_id` is a foreign key into the CDM `location` table — and the `location` table was populated with new `AutoGenId`-issued integers, not the source UUIDs. The source UUID was preserved in `location_source_value`.

So to set `person.location_id`, we look up in `location` by `location_source_value` and return the matching `location_id`. That's exactly what `GetCDMTableId` does — and it defaults to using `<table>_source_value` as the lookup key and `<table>_id` as the value, which is the convention we followed:



The arrow is `patients.csv:id` → `person.location_id`, with the rule:

```
GetCDMTableId(@location)
```

Because the rule references the CDM `location` table, the engine records an implicit dependency: `person` cannot be transformed until `location` has finished loading.

4.4.5 Recap

The `person` table now has:

- `Map(...) + AssertNot(0)` for each of `gender_concept_id`, `race_concept_id`, and `ethnicity_concept_id`, with the unmapped rows logged and skipped,
- `*_source_value` field-to-field arrows that preserve the original string,
- `birth_datetime` as a direct field-to-field arrow,
- `YearFromDate` / `MonthFromDate` / `DayFromDate` on the three integer date components,
- `AutoGenId` on `person_id`, with the UUID preserved in `person_source_value`,
- `GetCDMTableId(@location)` on `location_id`, which also expresses the load-order dependency.

Next: Useful patterns — vocabulary lookups against the OMOP standard vocabulary, Usagi maps, and conditional chaining.

4.5 Synthmas — clinical events

`visit_occurrence`, `visit_detail`, `condition_occurrence`, and `procedure_occurrence` all live in the OMOP CDM as records of something that happened to a patient at a point in time. They share a small handful of patterns:

- A fresh integer primary key from `AutoGenId`.
- A `*_type_concept_id` set with `SetValue`.
- Foreign keys (`person_id`, `visit_occurrence_id`, `visit_detail_id`, `provider_id`) resolved with `GetCDMTableId`.
- Vocabulary-coded concepts mapped via `VocabMap` + `VocabSourceId`.

Plus one pattern that's specific to `visit_occurrence`: `Selector` used as a type-based dispatcher.

4.5.1 The universal `*_type_concept_id`

Every clinical-event table has a `_type_concept_id` field that records *how* the row got there (EHR, claim, registry, derived…). For Synthmas, every event came from EHR-equivalent data, so they all get the same hard-coded value:

```
logic: SetValue(32810)         # "EHR" — visit, visit_detail, procedure, observation, …
logic: SetValue(32867)         # "EHR Chief complaint" — condition_occurrence
logic: SetValue(32817)         # "EHR prescription" — drug_exposure
```

The exact value comes from Athena — pick the concept id that best describes the provenance of your source data.

4.5.2 Cross-table foreign keys

Every clinical-event table has a `person_id` pointing back to `person`. The pattern is always the same — look up the source UUID in the already-loaded `person` table and demand a hit:

```
logic: GetCDMTableId(@person)
logic: AssertNot(NULL)
```

`GetCDMTableId(@person)` defaults to looking up `person_source_value` and returning `person_id`. The `AssertNot(NULL)` makes the missing-person case a logged failure rather than a silent NULL foreign key.

`visit_occurrence_id` and `visit_detail_id` follow the same pattern, but with a `NULL` default rather than a hard fail (a procedure may be unattached to a visit):

```
logic: GetCDMTableId(@visit_occurrence, NULL)
```

For `provider_id` on a child table, an explicit key/value form points the lookup at the right column:

```
logic: GetCDMTableId(@visit_occurrence, :visit_occurrence_source_value, :provider_id)
```

4.5.3 `visit_occurrence` — type-based dispatch with `Selector`

Synthmas's `encounters.csv` carries an `encounterclass` column that is *sometimes* an integer concept id and *sometimes* a class string (`ambulatory`, `emergency`, …). Two CDM fields draw from it differently:

```
visit_concept_id:
  logic: Selector(:encounterclass, "=", 1, :code, :encounterclass)
```

```
visit_source_value:
  logic: Selector(:encounterclass, "=", 0, :code, :encounterclass)
  logic: AsString()
```

Read the `Selector` calls as: *"if* `:encounterclass` *equals 1, return* `:code` *; otherwise return* `:encounterclass` *"*. So the engine picks between the source `code` and `encounterclass` columns depending on which side of the source data this row came from. The `AsString()` after the second `Selector` flattens the chosen value to a string for the source-value field.

4.5.4 Vocabulary-coded concepts (SNOMED, RxNorm, LOINC)

`condition_occurrence` , `procedure_occurrence` , and the other event-y tables all carry SNOMED-coded `code` columns in the source. The CDM wants both a standard concept id and a source concept id — the engine has rules for each:

```
condition_concept_id:
  logic: AsString()
  logic: VocabMap("SNOMED")

condition_source_concept_id:
  logic: AsString()
  logic: VocabSourceId("SNOMED")
```

`VocabMap` returns the *standard* concept id mapped from a vocabulary code (cross-walked through OMOP's mapping tables). `VocabSourceId` returns the OMOP concept id of the source code itself, without crosswalk.

For codes that aren't in the standard vocabulary — Synthea occasionally emits SNOMED codes that have been retired or replaced — a small inline `Map` ahead of `VocabMap` re-routes them:

```
logic: AsString()
logic: Map([
logic:   ["425525006", "782555009"],
logic:   ["300916003", "419412007"]
logic: ])
logic: VocabMap("SNOMED")
```

`Map` only fires for keys it recognises; everything else passes through unchanged to `VocabMap` . See Useful patterns → pre-map then vocabulary for variations on this theme.

4.5.5 Recap

The four event tables share:

- `AutoGenId` for the primary key.
- `SetValue(<concept_id>)` for `*_type_concept_id` .
- `GetCDMTableId(@person) + AssertNot(NULL)` for the mandatory FK.
- `GetCDMTableId(@visit_occurrence, NULL)` for optional FKs.
- `AsString() + VocabMap("SNOMED") + VocabSourceId("SNOMED")` for vocabulary lookups, with an optional pre- `Map` for out-of-vocabulary codes.

`visit_occurrence` adds the `Selector` -as-dispatcher pattern, which becomes a recurring trick once you start handling source data with multi-format columns.

Next: Values and dates — observations, measurements, and the date-arithmetic patterns for drug and device exposures.

4.6 Synthmas — values and dates

The observation/measurement/exposure tables are where the engine's value-handling and date-arithmetic patterns get a workout. Three techniques recur:

- **Mode-dispatch with `Selector`** — picking which source field is meaningful for this CDM field, based on a type or kind column.
- **Date arithmetic for end-dates** — when the source has a start date and a duration but no explicit end date.
- **Pipeline regex parsing** — extracting structured sub-fields out of a single source string (UDI codes).

4.6.1 `observation` — mode dispatch on `:type`

The OMOP `observation` table has separate fields for numeric and string values: `value_as_number` and `value_as_string`. The Synthmas source has one `value` column plus a `type` column that says which kind of value this row carries.

```
value_as_number:
  logic: Selector(:type, "=", "numeric", :value)

value_as_string:
  logic: Selector(:type, "=", "text", :value)
```

`Selector` here is a gate: "if `:type` equals 'numeric', return `:value`; otherwise return nothing (the field stays unmapped for this row)". Both fields look at the same source `value`, but only one is populated per row.

4.6.2 One source CSV, two CDM tables: how `observations.csv` splits

`observations.csv` is the source for both `observation` *and* `measurement`. The RiaH wires up two table-to-CDM arrows:

```
observations.csv ──► observation
observations.csv ──► measurement
```

Each arrow runs as its *own* mapping pipeline. Both pipelines read every row of `observations.csv`; what differs is which rows each one keeps. The `observation` pipeline uses the `Selector`-based gates shown above — text-typed rows land, numeric rows leave the value-fields empty. The `measurement` pipeline takes the opposite slice, but expresses it differently: a single rule on the chain that *rejects* non-numeric rows outright:

```
# On the observations.csv → measurement.value_as_number arrow:
logic: Assert("numeric")
```

`Assert` on a chain that's evaluated against the row's `:type` field throws when the source value isn't `"numeric"`. A thrown rule drops the whole row from the pipeline — the engine won't write a `measurement` record for it. The net effect: `observation` gets the text rows, `measurement` gets the numeric rows, both from the same input file.

The same shape — two arrows, two pipelines, one filter on each — shows up any time a source table needs to fan out by a category column. The thing to remember is that *both* pipelines read every source row independently; the engine doesn't optimise out the "already-routed" rows. That's why `measurement`'s `Records:` counter in `run.log` climbs noticeably higher than its final `Inserted` line — the Assert-rejected rows still get read and counted (see the FAQ).

4.6.3 `measurement.value_as_concept_id` — pick between two source fields

A different `Selector` pattern shows up on `measurement`: pick between *two* source value fields based on a description column.

```
value_as_concept_id:
  logic: Selector(:description, "=", "QOLS", :qols_value, :qaly_daly_value)
```

If the row's description is "QOLS", use the `:qols_value` field; otherwise use `:qaly_daly_value`. This is a clean way to handle source schemas where the meaningful column depends on which kind of record this is.

4.6.4 Filtering before vocabulary lookup

The same `measurement` field also wants the standard concept id mapped from a LOINC code in the source — but the source includes a few rows with synthetic codes (`QOLS` , `QALY` , `DALY`) that are not real LOINC. Filter them out before `VocabMap` :

```
measurement_concept_id:
  logic: AssertNot("QOLS")
  logic: AssertNot("QALY")
  logic: AssertNot("DALY")
  logic: VocabMap("LOINC")
```

The chained `AssertNot` s reject the synthetic codes per row (logging them but not failing the whole table), so only real LOINC values reach `VocabMap` .

4.6.5 `drug_exposure_end_date` — pipeline arithmetic with `Selector` fallback

The drug-exposure source has a `start` date and a `stop` date, but `stop` is sometimes null. When it's null, derive the end as `start + interval` , where `interval` is set elsewhere in the chain.

```
drug_exposure_end_date:
  logic: Add(:start, :interval) => :end_date
  logic: Selector(:stop, "<>", NULL, :stop, :end_date)
```

Read top-down:

1. `Add(:start, :interval) => :end_date` — `Add` sums a date and an integer-seconds interval into a new date. Because the chain uses the pipeline form `=> :end_date` , the result lands in the named binding rather than the chain's primary output. See DSL Guide → Pipelines.
2. `Selector(:stop, "<>", NULL, :stop, :end_date)` — if `:stop` is non-null, use it; otherwise use the computed `:end_date` .

This is the "computed default" pattern: prefer the source value when it's there, fall back to a derivation when it isn't.

4.6.6 `device_exposure_end_date` — fixed interval

Devices in Synthmas don't have a `stop` date at all, so the end is just start + a fixed 30 days. The new-mode parser supports **duration literals** (`30d` , `2h` , `1m` , …), which collapse the whole thing into one line:

```
device_exposure_end_date:
  logic: Add(:start, 30d)  // 30-day default exposure window
```

`30d` resolves to "30 days expressed as integer seconds" at parse time — no intermediate binding, no separate `Days()` call. The older shape

```
logic: Days(30) => :interval
logic: Add(:start, :interval)
```

still works in old mode. Use whichever reads more clearly to you; the duration-literal form is the easy readability win.

4.6.7 Device UDI parsing — pipeline regex extraction

Devices also have a UDI (Unique Device Identifier) column — something like `(01)0345678123(10)BATCH42(11)241015(17)270101(21)SN9876`. The CDM stores the UDI verbatim in `device_source_value`, but the engine can also pick out the named subfields with a series of `Regex` extractions:

```
device_source_value:
  logic: :udi => Regex(/\(10\)[0-9]+/) => :lot
  logic: :udi => Regex(/\(11\)[0-9]+/) => :date
  logic: :udi => Regex(/\(17\)[0-9]+/) => :exp
  logic: :udi => Regex(/\(21\)[0-9]+/) => :serial
  logic: Concat(:lot, :date, :exp, :serial)
```

Each `Regex` extracts one parenthesised UDI segment into a named binding (`:lot`, `:date`, `:exp`, `:serial`). `Concat` then joins the four into one string for the source-value field.

The standard concept id uses a different pattern — one regex, followed by `SubString`:

```
device_concept_id:
  logic: Regex(/\(01\)[0-9]+/)
  logic: SubString(4)
  logic: AsString()
  logic: VocabMap("SNOMED")
```

`Regex` returns the matched substring; `SubString(4)` drops the leading `(01)` prefix; `VocabMap` looks up the resulting bare GTIN as a SNOMED code.

4.6.8 `AsFloat` with a default

Numeric source fields that may be empty get an explicit default at the type cast:

```
value_as_number:
  logic: AsFloat(0.0)
```

Without the default, an empty string would fail the cast and skip the row. With `AsFloat(0.0)`, an empty string becomes `0.0`. See `AsFloat`.

4.6.9 Recap

The patterns introduced here:

- `Selector(:flag, "=", value, :a, :b)` — pick between fields based on a flag column.
- Chained `AssertNot` to filter values before a lookup.
- `Add(:date, :interval) => :end_date` — pipeline arithmetic, result into a named binding.
- `Days(N) => :interval` — duration literals stored in a binding.
- `Regex(/.../) => :name` — extract subfields with regex; `Concat` to recombine.
- Type casts with defaults (`AsFloat(0.0)`, etc.).

Next: Derived tables — `death` and `observation_period`, populated via `LeftJoin` and cross-table lookups rather than direct source-table mappings.

4.7 Synthmas — derived tables

Three CDM tables in this walkthrough aren't a 1-to-1 map of any source table. They're built from joins, cross-table lookups, or by drawing from two source tables at once — which is where the engine's table-level rules and multi-source row-construction earn their keep.

4.7.1 `death` — built from a `LeftJoin`

The OMOP `death` table needs a `person_id`, a `death_date`, and optionally a cause of death (a SNOMED concept). In Synthmas, the death date lives on `patients.csv` (column `deathdate`, often null) and the cause lives in `conditions.csv` (a condition row whose `start` matches the patient's `deathdate`).

We can't draw a single arrow for this — we need rows from both tables, joined by the matching date. That's a table-level `LeftJoin`, placed on the CDM `death` table's Comments field:

```
logic: LeftJoin(@patients.csv, @conditions.csv, [:id, :deathdate], [:patient, :start])
```

Read it as: *"left-join* `patients.csv` *and* `conditions.csv` *where the patient's* `(id, deathdate)` *matches the condition's* `(patient, start)` *".* `LeftJoin` keeps every patient row even when no matching condition exists — patients who haven't died still appear in the join result, just without a condition. The downstream `death_date is NULL` filter then drops them naturally (the field mapping skips rows that have no value).

Once the join is in place, the field-level rules on `death` look just like ordinary field-to-field mappings:

```
person_id:
  logic: GetCDMTableId(@person)
  logic: AssertNot(NULL)
death_date: <-- patients.csv:deathdate                    # field-to-field, no rule
cause_concept_id:
  logic: AsString()
  logic: VocabMap("SNOMED")                               # the condition's code
```

4.7.2 `observation_period` — cross-CDM-table lookup

The `observation_period` table records *the time span during which the engine's view of the person was valid.* There's no source table that captures it directly; it's derived from the source `patients.csv` file's earliest and latest event dates, with a fallback when the "latest" date is missing.

The interesting field is `observation_period_end_date`:

```
logic: Selector(:stop, "<>", NULL, :stop, :start)
```

If the source has a non-null `:stop`, use it; otherwise reuse `:start` (the period collapses to a single day, which is the right answer for a patient with no recorded follow-up).

The `person_id` field uses the by-now-familiar pattern:

```
person_id:
  logic: GetCDMTableId(@person)
  logic: AssertNot(NULL)
```

Because `GetCDMTableId(@person)` records an implicit dependency on the `person` table, the engine guarantees `person` is fully loaded before `observation_period` runs.

The table also declares a virtual field for traceability:

```
observation_period:
  field: observation_period_source_value
```

...and the `observation_period_id` rule writes into both the virtual field (via `SaveToVirtualField`) and the integer ID:

```
observation_period_id:
  logic: SaveToVirtualField(:observation_period_source_value)
  logic: AutoGenId()
```

Each row gets a unique CDM `observation_period_id` plus the source identifier preserved in the virtual field for downstream reference.

> **Automation alternative (Paid / AUTO)**
>
> This row-wise mapping is the Free-tier way to build `observation_period`. With a Paid licence carrying the **AUTO** feature you can instead let the engine derive it — one OMOP-correct period per person from the span of their clinical events — by setting `automation { post { create observation periods: true } }`. The same toggle also builds the `condition_era` / `drug_era` / `dose_era` tables, and all four now run on a **CSV target** as well as a database. A demo licence is available on request if you'd like to try it on this dataset.

4.7.3 `payer_plan_period` — two source tables, one CDM row

Synthea exposes the insurance picture across two files. `payer_transitions.csv` has one row per `(patient, payer, year-range)` — exactly the shape OMOP's `payer_plan_period` expects, so it drives row creation. `payers.csv` is the small lookup table that gives each payer UUID a human-readable `name` (`"Aetna"`, `"Medicare"`, `"NO_INSURANCE"`, ...).

The challenge is that the *row driver* and the *concept-id lookup source* aren't the same table. Just registering two table-to-CDM arrows isn't enough — the engine would run the two sources as **independent row generators**, producing one cohort of rows with patient/dates but no payer info, and another cohort with payer info but no patient/dates. To get a real composed row, the CDM table needs an explicit join:

```
# On the payer_plan_period CDM table Comments:
logic: LeftJoin(@payer_transitions.csv, @payers.csv, [:payer], [:id])
```

That tells the engine: *use* `payer_transitions` *as the driving table; for each driving row, look up the matching* `payers` *row on* `payer_transitions.payer = payers.id`*; combine both into the row that flows downstream into the field rules.* `LeftJoin` keeps the transitions row even if no payer match is found (in which case the payer-derived fields land as NULL); a plain `Join` would drop the row entirely.

With the join in place, the field rules split cleanly along the two source sides:

```
# From the payer_transitions side of the join:
patient     <-- payer_transitions:patient
              logic: GetCDMTableId(@person)
              logic: AssertNot(NULL)
start_year  <-- payer_transitions:start_year
end_year    <-- payer_transitions:end_year

# From the payers side of the join:
name        <-- payers:name
              logic: UsagiMap("payers.csv")        # → payer_concept_id
name        <-- payers:name
              logic: Truncate(50)                  # → payer_source_value
```

The `UsagiMap` looks up the payer name in `etc/etl-engine.d/usagi/payers.csv` and returns the matching SOPT concept id (`280` for Medicare, `289` for Medicaid, `327` for the named commercial payers, `436` for self-pay, and so on). The starter Usagi file shipped with the demo covers all ten payer names Synthea generates; extend or replace it for your own payer set.

The `payer_plan_period_id` field uses the same pattern you saw on `observation_period`:

```
payer_plan_period_id:
  logic: AutoGenId()
```

The other concept-id fields — `plan_concept_id`, `stop_reason_concept_id`, `plan_source_concept_id` — have no equivalent column in `payer_transitions.csv`, so they're left unmapped and default to NULL/0. That's the honest answer for Synthea's data; if your real source has plan and stop-reason fields, add the rules in your own RiaH.

The pattern this section introduces: *one CDM table fed by two source tables, where one drives the rows and the other contributes field values through an automatic foreign-key join*. The same shape shows up any time a fact table needs a label from a dimension table — common in real EHR ETLs.

4.7.4 Range joins — matching on more than equality

The joins above all match each key pair by **equality**. An optional **comparison array** — one operator per key pair (`"="`, `">"`, `">="`, `"<"`, `"<="`), AND-ed together like an SQL `ON` clause — lets a join match on order, not just equality:

```
# a.person_id = b.person_id AND a.date >= b.start AND a.date <= b.end
Join(@a.csv, @b.csv, [:person_id, :date, :date], [:person_id, :start, :end],
     ["=", ">=", "<="])
```

The `"="` pairs drive the fast index lookup; the comparison pairs filter the matched candidates — e.g. attaching the row whose `[start, end]` window contains an event date (an equality key on the person plus a date range).

A join with **no** `"="` pair at all is a *pure-range* join: with no key to index on, the right table is scanned and the comparisons do all the matching (useful for bucketing — e.g. a value falling between a lookup table's `low` / `high` bounds). Omit the operator array entirely and the join is plain equality, exactly as before. Comparison operands must be field references on both sides — a constant bound is a filter, not a join condition. `LeftJoin` takes the same operator array (every left row is still preserved; the comparisons decide which right rows match).

4.7.5 `cdm_source` — a single static row

The CDM `cdm_source` table holds metadata about *the data source itself* — typically one row, hard-coded. `InsertRecord` is the right tool:

```
logic: InsertRecord(
  cdm_source_name="Synthetic Massachusetts",
  cdm_source_abbreviation="Synthmas",
  cdm_holder="Rook IT ApS",
  source_description="Test translation of the Synthmas dataset",
  source_documentation_reference="https://mitre.box.com/shared/static/aw9po06ypfb9hrau4jamtvtz0e5ziucz.zip",
  source_release_date='2023-01-13',
  cdm_release_date='2023-06-30',
  cdm_version="5.4",
  cdm_version_concept_id=756265,
  vocabulary_version="v5.0 31-MAY-23")
```

Every non-nullable field must be supplied. The values are pure metadata — no source data flows in.

4.7.6 Splitting an ETL with `StoreIndexMap` *(Paid tier)*

For real ETLs that span more than one Rabbit-in-a-Hat file, the `StoreIndexMap` and `StoreSequence` table rules persist `(source-key → CDM-id)` maps and global sequences to disk. A second engine run, in a different Rabbit-in-a-Hat file, can then resolve the original source UUIDs back to the CDM integer IDs without re-loading either table.

```
location:
  logic: StoreIndexMap([:location_source_value], [:location_id])

visit_occurrence:
  logic: StoreIndexMap([:visit_occurrence_source_value], [:visit_occurrence_id])
```

These rules require a Paid licence — they're the foundation of multi-RiaH ETLs, which is firmly an enterprise pattern. The Synthmas walkthrough does not use them; everything in the bundled hat file is single-pass and works on the Free tier.

For more on the multi-RiaH pattern, see Folder Layout → Splitting an ETL.

4.7.7 Recap

`death`, `observation_period`, and `payer_plan_period` are all *derived* CDM tables — none of them comes from a single source table. The patterns they introduce:

- `LeftJoin` at the table level to construct a CDM table from two source tables matched on shared keys.
- Two table-to-CDM arrows pointing at the same CDM table, composed by an explicit `LeftJoin` at the CDM-table level so one source drives rows and the other contributes field values — *not* implicit; registering two arrows alone produces two parallel row streams.
- `GetCDMTableId(@person)` + `AssertNot(NULL)` for cross-CDM-table foreign keys with a hard-fail on miss.
- `Selector` for null-fallback (use `:stop` if non-null, otherwise `:start`).
- `UsagiMap` on a field value that came in via a joined-in source table — e.g. payer name from `payers.csv` joined onto a `payer_transitions.csv` row.
- `InsertRecord` for static metadata rows.
- `StoreIndexMap` for cross-RiaH ETL stitching (Paid tier; not in the bundled Synthmas RiaH).

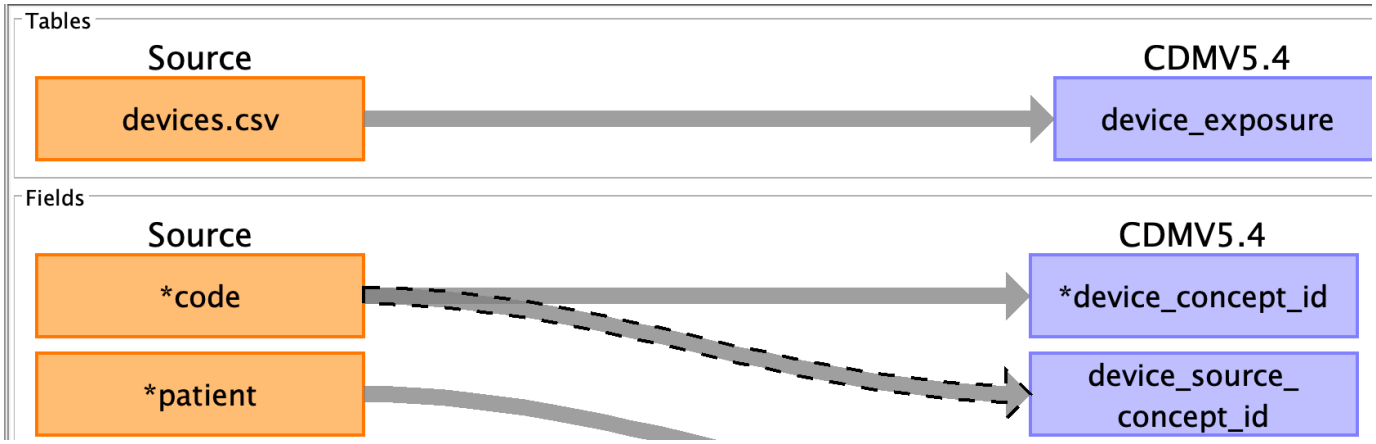
That's the whole walkthrough. The closing Useful patterns page collects the language-level techniques we built up — vocabulary lookups, Usagi maps, conditional chaining — in one reference for when you're writing a new mapping and want a reminder of the toolkit.

4.8 Synthmas — useful patterns

The location and person tables covered the bread and butter. This page collects a few patterns that come up repeatedly in real ETLs: vocabulary lookups against the OMOP standard vocabulary, Usagi-based mappings for non-vocabulary terms, and conditional chaining when several lookup strategies need to be tried in priority order.

4.8.1 Pattern 1 — vocabulary lookups for known codes

Synthmas's `devices.csv` has a `code` column with SNOMED codes identifying each device (often a defibrillator). We want the CDM `device_concept_id` to hold the OMOP standard concept id for that SNOMED code, and `device_source_concept_id` to hold the OMOP concept id for the SNOMED code itself.



The SNOMED code is stored as an integer, so we cast to string first, then look it up:

For `device_concept_id` (the standard concept id):


```
AsString()
VocabMap("SNOMED")
```

For `device_source_concept_id` (the original SNOMED concept id):


```
AsString()
VocabSourceId("SNOMED")
```

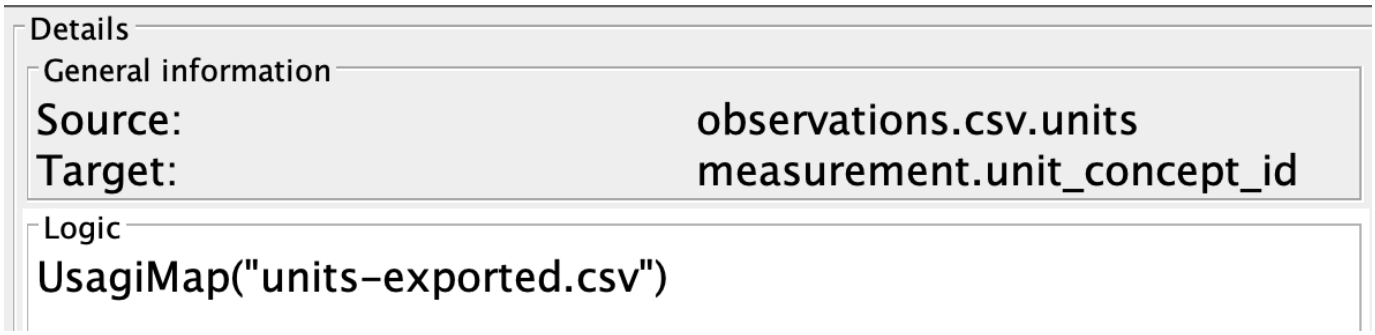
`VocabMap` returns the standard concept id mapped from a vocabulary code; if the SNOMED code maps to a non-SNOMED standard, you get the cross-walk for free. `VocabSourceId` returns the OMOP concept id of the source code itself, which lets the CDM remember "this came from SNOMED `12345` " without having to store the SNOMED string.

4.8.2 Pattern 2 — Usagi for codes the vocabulary doesn't have

Sometimes the source data has codes (or, more often, free-text strings) that aren't in any standard OMOP vocabulary. The OHDSI solution is Usagi: a tool that helps a domain expert curate a CSV mapping from source strings to OMOP concept ids.

Synthmas's `observations.csv` has a `units` column with strings like `"mg/dL"`. The CDM `observation.unit_concept_id` is an integer — so those strings need a Usagi map.

The Usagi process produces a CSV file with columns like `sourceCode` , `targetConceptId` , etc. Drop it into `etc/etl-engine.d/usagi/` and reference it with `UsagiMap` :




```
UsagiMap("units-exported.csv")
```

`UsagiMap` is also the right tool when the source string lives on a different table from the CDM row being built — see `payer_plan_period` in the derived-tables page for the multi-table-join version (`payer_transitions.csv` drives the rows, `payers.csv` is brought in by an explicit `LeftJoin` at the CDM-table level to provide the `name` string the Usagi lookup uses).

4.8.3 Pattern 3 — pre-map then vocabulary

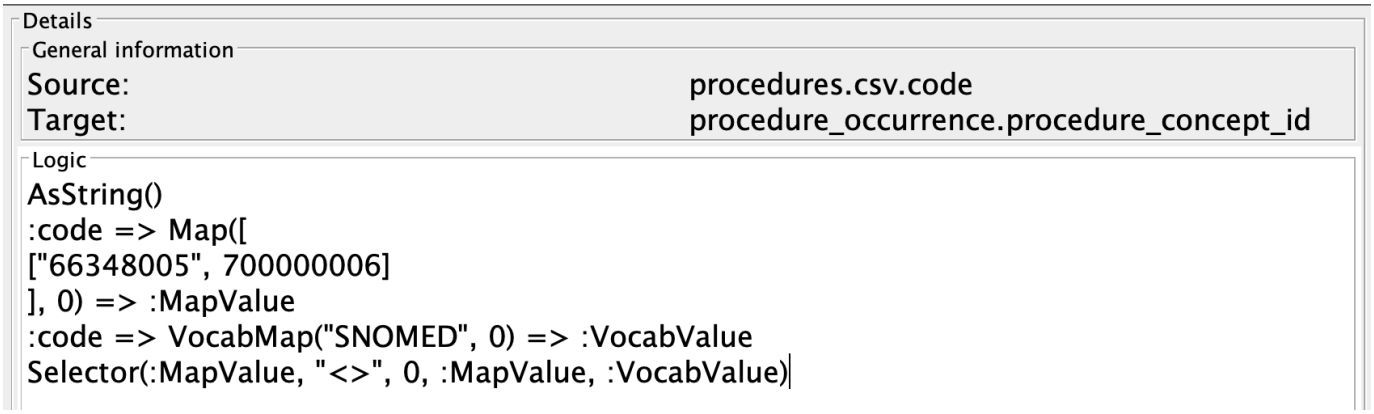
Sometimes a small set of source codes are missing from the vocabulary but you know what they should map to. The cleanest fix is to add a `Map` *before* the `VocabMap`, replacing the missing codes with codes the vocabulary does have:


```
Map([
  ["legacy_code_A", "ICD10_A12.3"],
  ["legacy_code_B", "ICD10_B45.6"]
])
VocabMap("ICD10")
```

The pre-map only fires for the listed values; everything else passes through to `VocabMap` unchanged. If the pre-map's output doesn't have to itself live in the same vocabulary as the `VocabMap` argument, you've reached the limit of this pattern — use conditional chaining (below) instead.

4.8.4 Pattern 4 — `Map` with `Selector`, the verbose form

When the pre-map approach doesn't fit, you can run two independent maps into separate variables and pick whichever one succeeded:




```
:source_code => Map([..., 0])      => :hand_curated
:source_code => VocabMap("ICD10")  => :vocab
Selector(:hand_curated, "!=", 0, :hand_curated, :vocab)
```

This works, but it's verbose, and the `Selector` line is easy to get wrong. Conditional chaining is shorter and faster.

4.8.5 Pattern 5 — conditional chaining (`||`)

The same logic as pattern 4, written with `||`:


```
Map([
  ["legacy_code_A", 12345],
  ["legacy_code_B", 67890]
]) || VocabMap("ICD10")
```

If the first rule fails to produce a mapping (the default is `null` unless you supply one), the second runs against the same input. The chain can be any length:

```
Map([...]) || UsagiMap("file.csv") || Map([..., default_code])
```

The final entry can be a `Map` with a default value as a catch-all that guarantees a successful mapping no matter what came in.

See DSL Guide → Conditional Chaining for the full mechanics — in particular, which rules support `||` chaining (mapping rules: most do; aggregating rules: only some).

4.8.6 What's next

You now have the toolkit:

- table mapping with `Normalize`, `Join`, `DependOn`, `InsertRecord`,
- field mapping with type casts, padding, defaults, asserts, and date deconstruction,
- `AutoGenId` and `GetCDMTableId` for ID management,
- `VocabMap` / `VocabSourceId` / `UsagiMap` for vocabulary lookups,
- conditional chaining with `||` to combine them.

For more on any individual rule, see the Rule Reference. For the full language grammar, see DSL Guide → Mapping Language Syntax.

If you have a worked example of your own that's worth writing up, follow the same shape as Synthmas: an `index.md` overview, one page per CDM table that's interesting, and a closing useful-patterns page. See Examples → Adding new examples.

4.9 Synthmas — FAQ

Questions that come up when you run the bundled Synthmas demo for the first time and watch `logs/run.log` or compare row counts in `etc/etl-engine.d/cdm-out/`. Most of these aren't bugs — they're artefacts of how the engine reports progress versus what it actually persists.

4.9.1 The `Records:` counter in `run.log` climbs past 150,000 on the Free tier — isn't there a cap?

There is — and it's holding. What you're seeing in `run.log` is the counter for *source rows the mapper has processed* (`rowsHandled`), not *rows committed to the CDM table* (`rowsCommitted`). The Free tier cap is on the committed side: once a CDM table has accepted 150,000 rows, further rows are silently dropped at the persistence step, but the upstream reader keeps pulling source rows and the mapper keeps chewing through them.

The dropped rows show up in two places:

- The cap was reached. A one-shot warning is written to both `run.log` and the per-table log (e.g. `logs/measurement.log`): `Row cap reached for measurement (150000) — additional rows dropped`.
- The output CSV stops growing. After the cap, `cdm-out/<table>.csv` has exactly 150,000 data rows even if `Records:` ended at 500,000.

Once the cap fires, the engine also stops reading further source rows for that table (in newer builds) — so for *unfiltered* tables the gap between the final `Records:` value and 150,000 is small.

For tables whose rule chain *filters* rows mid-pipeline, the gap can still be large. In the bundled Synthmas demo, `measurement` illustrates this — its source is `observations.csv` (~531 K rows), of which ~60 % carry `TYPE=numeric` and ~40 % carry `TYPE=text`. The RiaH puts an `Assert("numeric")` on the chain so text rows are dropped before insert. The cap fires on commit #150,000, but to *get* 150 K numeric commits, the engine had to chew through roughly 250 K source rows (150 K commits + ~100 K text rejections). The in-flight tail at cap-fire then runs another ~25 K rows before the upstream short-circuit lands. End result in a recent run:

```
[measurement] Handled 274,442 records
[measurement] Inserted 150,000 records
```

That's not the cap leaking; it's the cap counting commits while `Records:` counts everything the mapper has looked at. Tables with no in-chain filter (like `procedure_occurrence`) land at `Handled 150,030 / Inserted 150,000` instead — the small tail is just the in-flight rows at cap-fire moment.

See the next entry for the underlying handled-vs-written distinction that makes this kind of mismatch possible in the first place, and Licensing → Free for the tier overview.

4.9.2 Why doesn't `Records:` in `run.log` match the row count in my CDM CSV?

It almost never will, even with no cap in play. `Records:` is the *handled* count: the number of source rows the mapper has pulled in and run through the rule chain for the current CDM table. The CDM-side row count is the *written* count: how many rows actually landed in `cdm-out/<table>.csv` (or in the target database table on Paid tier). They're two different quantities, and the gap between them is normal — the rules in your RiaH actively *change* how many output rows each input row produces:

- **Fan-out rules write more than one CDM row per source row.** Multi-result mappings like `Map(...)[]` or unrolled `LoopOver` rules emit several output values for a single input row. The classic Synthea→OMOP case: a `claims.csv` row with eight diagnosis columns fans out to up to eight `condition_occurrence` rows. The written count climbs faster than the handled count.
- **Filtering rules write fewer than one CDM row per source row.** Anything that returns "no value" for the row — a `Selector` that picks a NULL branch, a conditional chain (`||`) where every branch fails, a `LeftJoin` row with no match on the join key, a `Normalize` that deduplicates against an earlier row — drops the row at the persistence step. The written count climbs slower.
- **Both, in the same table.** Real mappings combine these. The `Records:` counter doesn't reflect either.

The Free-tier 150,000 cap applies to the *written* count, not the handled count — which is why the previous entry's "`Records:` past 150,000" symptom happens. But the same mismatch happens at any scale, with or without a licence: if you cross-check `run.log` against `wc -l cdm-out/<table>.csv` you should *expect* them to differ.

When a CDM table finishes, the engine writes both numbers to the per-table log (`logs/<table>.log`):

```
[procedure_occurrence] [info] Handled 235460 records
[procedure_occurrence] [info] Inserted 150000 records
```

`Handled` is what `run.log` was tracking; `Inserted` is what landed in `cdm-out/<table>.csv`. The same per-table log shows `skipping row` warnings for individual filter / no-mapping / no-match cases while the table is running, which is how you'd track filtering live — though there's no live counter for the cumulative committed total until the summary line at the end.

4.9.3 I tried an RDBMS target, automation, or `StoreIndexMap` and the run stopped with an error

Those are Paid-tier features, and the Free tier surfaces them loudly rather than ignoring them silently: the engine stops with a clear message at config-parse or table-load time, naming the feature and the licence flag it needs (e.g. `OPEN_SOURCE_RDBMSES`, `AUTO`, `ETL`). To keep running on Free, drop the feature from your `etl.conf` / RiaH; otherwise add a Paid licence — see Licensing.

4.9.4 Throughput swings from 10 rec/s to 10,000 rec/s mid-table — is that normal?

Yes, and it's usually one of three things, sometimes all at once:

- **Downstream queue backpressure.** When the writer can't keep up (e.g. CSV writer holding its mutex with a per-row flush, or RDBMS batch inserts catching their breath), the mapper's output queue fills to its cap and the producer yields. As soon as the writer drains a batch, throughput jumps. The `Output queue: 100` line you see in `run.log` for several seconds in a row is this happening.
- **Per-row cost variance.** Multi-result fan-outs, joins, conditional chains, and `Selector` rules don't all cost the same. A row that hits a fast path runs faster than one that triggers a vocab lookup *and* a Usagi join *and* a fan-out.
- **Cold-cache vocab loads.** When a row references a vocabulary concept that hasn't been touched yet on this run, the engine fetches it from the loaded `CONCEPT.csv`. With the CSV CDM driver there's no index — concept lookups scan, which is slow until the in-memory cache is warmed. The first few rows of a new source table mid-run can stall hard if they pull in a cluster of previously-unseen concepts; the same rule on the same table runs 100× faster once the cache is warm.

See Troubleshooting → Performance for the deeper treatment of all three.

4.9.5 Is ~10,000 rec/s the ceiling?

Not at all. Depending on hardware and mapping complexity, the engine will reach **100,000+ rec/s** on tables with cheap rules and a warm cache. The Synthmas demo's `~10k rec/s` plateau on `measurement` and `observation` is a Free-tier, single-thread, CSV-target combination running through joins, vocab lookups, and Usagi joins — slow side of the design space. Real-world ceilings move with:

- Worker thread count (Paid tier removes the single-thread cap).
- Target driver (RDBMS batched inserts beat per-row CSV writes).
- Rule cost (a plain field-to-field copy is much cheaper than a multi-rule chain).
- Vocabulary cache size (`vocabulary cache size:` in `etl.conf`).

Whatever number you see in your `run.log`, treat it as a property of *this mapping on this machine* — not a published spec.

4.9.6 `Input queue` and `Output queue` show the same number every line — bug?

Cosmetic, and the labelling will improve in the next engine iteration. What `run.log` currently prints is the producer's output-side queue depth under both labels, because a producer→shared-queue→consumer pipeline sees the same physical queue from both ends. The number itself is meaningful (it's the live backlog the mapper is feeding the writer); the duplication is just the label aliasing.

For tables that use an asymmetric two-queue topology (`measurement` , `visit_detail` , the final `condition_occurrence` pass, `cost`) the two columns *do* diverge — typically `Input queue: 0 - Output queue: 100` , meaning the reader is keeping up and the writer is the bottleneck. That diagnostic does work today; it's the symmetric-queue case where the labels are confusingly tautological.

4.9.7 `procedure_occurrence` "loaded 205,000 records" but the CSV has 150,000 rows — what happened?

Two effects compounded:

- **Source-row fan-out.** One source row can produce more than one CDM row. A single Synthea `procedures.csv` row that maps via a multi-result chain ends up writing several `procedure_occurrence` rows. So `205,000` is the *source* row count the mapper handled; the corresponding CDM row count produced (pre-cap) was higher.
- **Free-tier cap.** Of the CDM rows the mapper produced, only the first 150,000 were committed; the rest were dropped. See the first question on this page for the full mechanics.

Both numbers are correct. They measure different things.

4.9.8 I added two source tables to one CDM table and half my rows are missing fields — what happened?

Most likely you registered two table-to-CDM arrows (or relied on both tables already being registered) and assumed the engine would auto-join them on a foreign key. It doesn't — two arrows without an explicit join rule produce **two independent row streams**. You'll see one cohort of rows with fields populated from source A and the source-B fields all NULL, plus a second cohort with the opposite shape.

Sanity check: count distinct values of a field that *only* the joined-in source could populate. In the `payer_plan_period` case, before the fix the output looked like this:

```
$ awk -F, 'NR>1 {print $1}' cdm-out/payer_plan_period | sort | uniq -c
      1 280       # 1 row per payer name in payers.csv
      1 288
      1 289
      6 327
      1 436
  51794 null      # 51,794 rows from payer_transitions, no payer
```

That `null` plateau is the tell: the dimension-table feed (`payers.csv`) produced exactly one row per source row, while the fact-table feed (`payer_transitions.csv`) produced ~50K rows with no payer data attached.

Fix: declare the join explicitly on the CDM table's Comments field:

```
logic: LeftJoin(@payer_transitions.csv, @payers.csv, [:payer], [:id])
```

The driving table goes first; the lookup table goes second; the shared key is the last two list arguments (`@driving.key` vs. `@lookup.key`). Use `LeftJoin` if you want every driver row to survive even when the lookup miss s; use `Join` if missing lookups should drop the driver row.

The same trap can hit any composed CDM table — for example a fact table that joins against a code-lookup CSV to enrich a `*_source_value` . Two arrows alone won't do it; the join rule has to be there.

4.9.9 `payer_concept_id` is 0 / 100% unmapped in `payer_plan_period` — what's missing?

If you're running the bundled demo as shipped, this *shouldn't* happen — the bundle includes a starter `etc/etl-engine.d/usagi/payers.csv` covering the ten payer names Synthea generates (Aetna, Medicare, Dual Eligible, NO_INSURANCE, …) and the RiaH wires `payers.name → payer_concept_id` through `UsagiMap("payers.csv")`. Check three things, in order:

1. **The Usagi file is present.** `ls etc/etl-engine.d/usagi/` should list `payers.csv` alongside `units-exported.csv` and `specialities.csv`. If `payers.csv` is missing, the RiaH still runs but the lookup returns 0 for every row.
2. **`etl.conf` points at the Usagi directory.** The `rabbit configuration` block in `etl.conf` should set `usagi directory: "/etc/etl-engine.d/usagi/"`. Without it the engine ignores the directory.
3. **The Usagi file has every payer name in your data.** Synthea is deterministic about the ten names it emits, but if you replaced the source CSVs with your own Synthea run that uses a custom payer module, names outside the starter map will resolve to 0. The fix is to add rows to `payers.csv`.

For payer concept choices in the starter map (which SOPT concept maps to which Synthea name), see the derived-tables walkthrough.

4.10 Patterns

Self-contained recipes for the engine's flow-control features. Where the Synthmas walkthrough follows one dataset end-to-end, this section is the inverse: one feature per page, with a small focused example that exercises it.

4.10.1 In this section

- Multi-result mappings — `Rule()[]`, `LoopOver`, result-index dispatch, and how multi-result rules fan a source row into several CDM rows.
- Conditionals (If / Else / Switch / Case) — branching inside a rule chain, the implemented syntax, when to reach for it.
- Fallback chains (`||`) — try-this-then-that across *blocks* of rules, not just single rules.
- Parallel collection (`++`) — run independent blocks against the same input and collect every result.

4.10.2 Where the syntax lives

All four patterns are part of the **new parser** (1.4 and later). The older single-rule `||` continues to work unchanged — see DSL Guide → Conditional Chaining. The block forms (`{ ... }(:var) || { ... }(:var)` and the `++` operator) and the `If` / `Switch` keywords are the new additions.

Every example below was lifted from a real test in the engine's test suite — file pointers are at the bottom of each page. If a test name is mentioned, the directory at `tests/<name>/` in the repo contains the full Rabbit-in-a-Hat fixture.

4.11 Multi-result mappings

A multi-result rule produces more than one output value per input row. The engine fans the source row into one CDM row per result — this is the standard pattern for "one source observation maps to several OMOP concepts" and similar 1-to-N transformations.

4.11.1 The `[]` and `[N]` suffixes

The suffix tells the parser the rule is multi-result.

Suffix	Meaning
`[]`	Variable number of results — produce as many rows as matches.
`[N]`	Exactly `N` results expected. Fewer → row is skipped + logged.

```
UsagiMap("file.csv")[]    # variable
UsagiMap("file.csv")[3]   # exactly 3
```

4.11.2 `LoopOver` declares how multi-results combine

When several CDM fields all carry multi-result rules, the engine needs to know whether to **zip** them (i-th of one paired with i-th of another) or take the **cartesian product**. `LoopOver` on the CDM table is what declares this:

```
logic: LoopOver([:field_one, :field_two])           # zip
logic: LoopOver([:field_one], [:field_two])         # cartesian
logic: LoopOver([:field_one], [:field_two], [:f3])  # 3-way cartesian
```

Within a single bracketed group, all fields are paired by index — they must produce the same number of results at runtime. Across groups, every combination is generated.

The `all=` keyword pulls in fields that aren't multi-result but should still be carried into every output row:

```
logic: LoopOver(all=[:person_id], [:year_of_birth, :birth_datetime], [:month_of_birth])
```

`:person_id` is single-valued and rides along; the `[:year_of_birth, :birth_datetime]` group is zipped (i-th year paired with i-th datetime); the `[:month_of_birth]` group joins as a cartesian factor.

4.11.3 Worked example — multi-result `UsagiMap` with dependent ID generation

Source: `intnum.csv` with four integer columns. Goal: map them through a Usagi file (which can return multiple matches per input) and let the CDM generate `person_id` values that follow the multi-result of `year_of_birth`.

CDM `person.year_of_birth`:

```
logic: UsagiMap("B.csv")[]
logic: AssertNot(21)
```

The `[]` says "every match becomes a result". `AssertNot(21)` runs on each result — any result equal to 21 fails that path, leaving only the surviving years.

CDM `person.person_id`:

```
logic: AutoGenId([:year_of_birth, :month_of_birth])[]
```

`AutoGenId` takes a list of dependency fields. With `[]` it produces one new id per combined-result row — the engine sees that `:year_of_birth` is multi-result and re-dispatches `AutoGenId` accordingly.

CDM `person.birth_datetime` (computed from the multi-result year):

```
logic: GetCDMFieldValue(:year_of_birth)[] => :year
logic: Add(1900) => :year
logic: MergeDate([:year, "year"])
```

`GetCDMFieldValue(:year_of_birth)[]` reads the multi-result of another field on the same CDM table — this is the cleanest way to keep two fields' multi-results in lockstep.

CDM `person` (table-level):

```
logic: LoopOver(all=[:person_id], [:year_of_birth, :birth_datetime], [:month_of_birth])
```

`person_id` is generated once per output row; `year_of_birth` and `birth_datetime` are zipped (index-paired); `month_of_birth` factors in as a cartesian dimension.

4.11.4 Result-index dispatch (block syntax)

When a multi-result rule produces a fixed number of results and each result deserves a different downstream chain, the block syntax dispatches by index:

```
SomeRule()[3] => {
  Rule_A1()
  Rule_A2()
}, {
  Rule_B1()
}, {
  Rule_C1()
  Rule_C2()
  Rule_C3()
}
```

Result `[0]` enters block 1, `[1]` block 2, `[2]` block 3. With `[]` (variable count) the same block runs against every result.

4.11.5 Caveats

- **Empty results skip the row.** `Rule()[]` that returns zero matches drops the source row — log it with an `AssertNot` or a default if you need visibility.
- **`LoopOver` is mandatory** when a CDM table has any multi-result field, even if there is only one such field.
- **`Spawn(:array)` is the new-mode replacement** when the multi-result is collected as an `ARRAY` value rather than fanned out immediately. See Arrays and New Mode.

4.11.6 See also

- `UsagiMap` — the most common multi-result rule.
- `AutoGenId` — multi-result-aware ID generator.
- `LoopOver` — declares how multi-results combine.
- DSL Guide → Multi-Sized Results.

4.11.7 Source tests

`tests/test062/` — multi-result `UsagiMap`, `AutoGenId` with dependencies, `GetCDMFieldValue`, the `all=` form of `LoopOver`.
`tests/test081/` — multi-result `Use()[N]` and `AutoGenId()[N]`.

4.12 Conditionals — `If` / `Else` / `Switch` / `Case`

The new parser adds proper structured conditionals, replacing the low-level `JumpIf` / `JumpIfNot` / `Mark` / `JumpTo` cluster for the common cases.

4.12.1 `If` / `Else`

The basic shape:

```
If (<condition>) {
  ... rules to run when condition is true ...
} Else {
  ... rules to run when condition is false ...
}
```

The condition is an infix expression — comparisons return booleans, which the `If` consumes. Both branches are blocks of rules; either branch may be omitted (`Else` is optional). Variables modified inside a branch persist into the rest of the chain.

Worked example — fallback when a primary lookup is null

Source `intnum.csv` has four integer columns. The primary lookup sometimes returns `NULL`; in that case, fall back to a different source field through a different Usagi map.

```
logic: :int1 => UsagiMap("A.csv", NULL)[] => :var1
logic: If (:var1 = NULL) {
logic:   :int4 => UsagiMap("D.csv", NULL)[] => :var1
logic: }
```

Read top-down:

1. Run `UsagiMap("A.csv")` against `:int1`, defaulting to `NULL` on a miss. Store the multi-result in `:var1`.
2. If `:var1` is `NULL`, replace it with the result of running `UsagiMap("D.csv")` against `:int4`.

Note `=` is the comparison operator (not `==`).

When to use `If` vs `||`

The single-rule `||` is shorter when both branches are one rule:

```
:int1 => UsagiMap("A.csv") || UsagiMap("D.csv")
```

`If` wins when:

- A branch has **multiple rules** to execute as a unit.
- The branches operate on **different inputs** (the `||` form requires the same input for both alternatives).
- A condition is **expressed by comparison** rather than by "did the rule succeed".

For multi-rule branches that share an input, the block fallback form `{ ... }(:var) || { ... }(:var)` is also worth considering — it keeps the "first success wins" semantics of `||` while letting each branch be a block.

4.12.2 `Switch` / `Case`

A clean cascade for many small lookups against a single value:

```
logic: Switch (:intnum2) {
logic:   Case 1: SetValue(38003563)
logic:   Case 2: SetValue(38003564)
logic:   Case 3: SetValue(38003565)
logic:   Default: SetValue(0)
logic: }
```

Equivalent to a chain of `If` / `ElseIf`s, but reads more naturally when the cases are all literal values against the same expression.

Worked example — concept-id by enum

Source `intnum1` carries a small set of enum-style integers. The target CDM field `gender_concept_id` is a real OMOP concept id, so each enum value maps to a different concept:

```
logic: Switch (:intnum1) {
logic:   Case 999: SetValue(9999)
logic:   Case 100: SetValue(8507)
logic:   Case 200: SetValue(8532)
logic:   Default: SetValue(0)
logic: }
```

For string keys this is essentially what `Map([...], default)` does in one line. `Switch` is the better choice when the right-hand side of each branch is a *block of rules* rather than just a value.

4.12.3 How conditionals desugar

Under the hood, `If` / `Else` and `Switch` / `Case` compile down to the `JumpIf`, `JumpIfNot`, `JumpTo`, and `Mark` rules. You can write those directly if you need control flow that the structured forms do not express, but the structured forms are strongly preferred — they're easier to read and harder to get wrong.

4.12.4 See also

- `JumpIf` and `Mark` — the underlying primitives.
- Conditional Chaining — `||` for single-rule fallback.
- Fallback chains (`||`) — block-level fallback.

4.12.5 Source tests

`tests/test062/` — `If` for null-fallback after a multi-result `UsagiMap`. `tests/test067/` — `Switch / Case` for enum-to-concept-id mapping.

4.13 Fallback chains (||)

The single-rule `||` operator (`Rule1() || Rule2()`) extends to whole **blocks** of rules. Block-level fallback is the right tool when the fallback path needs more than one rule — e.g. lookup, normalise, assert, all as a unit.

4.13.1 Syntax

```
{ Rule_A1() Rule_A2() }(:result) || { Rule_B1() Rule_B2() }(:result)
```

Each block declares its output with `(:var)`. All blocks in the chain must declare the same variable, with the same type — the parser checks this at compile time.

4.13.2 Semantics

- Blocks are evaluated **left to right**.
- The first block that **produces a value for** `:var` wins. Remaining blocks are not evaluated.
- A block **fails** if an `Assert` / `AssertNot` fires inside it. The chain then moves to the next block.
- `successfulMapping` is `true` if any block succeeded.

The single-rule form `Rule1() || Rule2()` is shorthand for `{ Rule1() }(:implicit) || { Rule2() }(:implicit)`. Both behave identically — the block form just lets you put more than one rule in each path.

4.13.3 Worked example — try one Usagi map, fall back to another

CDM `person.gender_concept_id`. Two Usagi files exist:

- `map3.csv` — a curated, high-precision file. May not have a match.
- `map4.csv` — a broader fallback with a default of `8521` ("UNKNOWN") so it always produces *something*.

We want the precise mapping where possible; otherwise the broad one.

```
logic: { UsagiMap("map3.csv")[] }(:gender)
logic: || { UsagiMap("map4.csv", 8521)[] }(:gender)
```

Read it as: try `map3` first; if no match, try `map4` (which has a default and so always succeeds). Both paths produce a multi-result into the same variable `:gender`, which the table-level `LoopOver` then expands into one row per match.

4.13.4 Multi-rule block fallback

Where the fallback needs *more than one* rule, the block form becomes essential:

```
{
  :raw_code => Map([..., NULL])
  AssertNot(NULL)
}(:concept_id)
||
{
  :raw_code => UsagiMap("vocab-map.csv")[]
  AssertVocabDomain("Condition", TRUE)
}(:concept_id)
```

If the hand-curated `Map` either misses the input or produces `NULL`, the `AssertNot(NULL)` fires and the first block fails — at which point the Usagi-based block runs and validates that the result is in the expected domain.

4.13.5 Type uniformity

All blocks must agree on the type of the declared output. The compile-time check catches mistakes like:

```
{ Map([["A", 1]]) }(:x)         # produces INT
|| { Map([["B", "fallback"]]) }(:x)   # produces STRING — error
```

The mismatch is rejected at parse time.

4.13.6 Single-rule `||` is unchanged

```
Rule1() || Rule2()
```

still works exactly as before — see DSL Guide → Conditional Chaining for the original semantics. Block-level `||` is purely an extension.

4.13.7 See also

- Conditional Chaining — the single-rule form.
- Conditionals — when `If` / `Else` is a better fit (different inputs, comparison-driven branching).
- Parallel collection (`++`) — the dual: collect every block's result instead of just the first success.

4.13.8 Source tests

`tests/test089/` — block fallback between two `UsagiMap` calls; the second has a default value so it always succeeds. `tests/test090/` — variations with named variables and multi-result.

4.14 Parallel collection (++)

The dual of `||` : where `||` runs blocks until one succeeds, `++` runs **every** block and **collects every surviving result** into an array.

4.14.1 Syntax

```
{ Rule_A1() Rule_A2() }(:result)
++ { Rule_B1() }(:result)
++ { Rule_C1() Rule_C2() }(:result)
```

Each block declares its output with `(:var)`. All blocks must use the same variable name and produce the same value type — checked at compile time.

4.14.2 Semantics

- All blocks are evaluated **independently**, each starting from the same input state.
- Variables modified inside a block stay inside that block — there is no leakage between paths.
- Each block that produces a value for `:var` contributes to the result.
- Blocks that **fail** (an `Assert` fires) contribute nothing and are silently dropped — failure of one path does not abort the others.
- The output of `++` is always `Array(T)` , where `T` is the declared type of `:var` .

Result flattening

Outputs are flattened one level:

Path A produces	Path B produces	`++` output
scalar `1`	scalar `2`	`Array(Int) = [1, 2]`
`[1, 2]`	scalar `3`	`Array(Int) = [1, 2, 3]`
`[1, 2]`	`[3, 4]`	`Array(Int) = [1, 2, 3, 4]`

So a path that internally produces multi-results — typically through `UsagiMap()[]` — contributes each of those individually to the collected output. `Array(Array(T))` never occurs for scalar/array outputs. Tuple outputs are kept as `Array(Tuple(...))` to preserve the correlation between fields.

4.14.3 Worked example — concepts from two parallel maps

Same `gender_concept_id` field as the fallback example, but here we want **every** concept any map can offer — both the precise and the broad — and let downstream rules deduplicate.

```
logic: { UsagiMap("map3.csv")[] }(:gender)
logic: ++ { UsagiMap("map4.csv")[] }(:gender)
```

Read it as: run `map3.csv` and `map4.csv` against the same input, keep all matches from both, deliver a single flattened `Array(Int)` containing every match.

The CDM-table-level `LoopOver([:person_id, :gender_concept_id])` then expands the array into one row per match.

4.14.4 When to reach for `++` vs `||`

- `||` — *prefer* one path; only fall back to the next if the preferred fails. Used when one path is canonical and others are backups.
- `++` — every path is *equally valid*; the union of their results is the answer. Used when source data has multiple valid interpretations or when you want to apply several mappings without picking a winner.

4.14.5 When `++` is overkill

If the parallel paths produce **scalar** results that you want combined with arithmetic — e.g. compute several values and add them — you do not need `++`. Just write the rules sequentially and combine with `Add`, `Selector`, etc. `++` is specifically for the "collect array of results" case.

4.14.6 Restrictions

- `++` blocks **cannot directly contain another** `++`. Indirect containment (e.g. inside an `If` branch within a `++` block) is fine.
- `||`, `If`, and multi-result rules (`UsagiMap()[]`) are all valid inside a `++` path.
- All blocks must declare the same output variable with the same type.

4.14.7 See also

- Fallback chains (`||`) — the dual of `++`.
- Multi-result mappings — how `Array(T)` outputs fan into rows via `LoopOver`.
- Arrays and New Mode — first-class `ARRAY` type that `++` produces.

4.14.8 Source tests

`tests/test091/` — basic `++` between two `UsagiMap` blocks.

5. Reference

5.1 Rule Reference

5.1.1 Rule reference

The ETL Engine ships with **102** documented mapping-language rules, grouped by family.

- Aggregating rules — 50 rules
- Control rules — 5 rules
- Filter rules — 2 rules
- Mapping rules — 36 rules
- Table rules — 9 rules

5.1.2 Mapping Rules

Mapping rules

Mapping rules transform a single value into a single value.

ARITHMETIC

- HighInt — Maps a floating point value to an integer by rounding the value up.

ARRAY

- At — Returns the element of an `ARRAY` at the given 0-based index as a scalar.
- InsertAt — Inserts a scalar value at the given 0-based index of an `ARRAY`, returning a new `ARRAY`.
- Remove — Drops every element of an `ARRAY` equal to the given scalar value.
- RemoveAt — Removes the element at the given 0-based index from an `ARRAY`.
- ReplaceAt — Replaces the element at the given 0-based index of an `ARRAY` with a new value.

ASSERTION

- Assert — Fails the record unless the input matches a value or one entry of an array of values.
- AssertNot — Fails the record when the input matches a value or one entry of an array of values.
- AssertVocabDomain — Fails the record unless the input concept id belongs to a named OMOP vocabulary domain.

CDM-HELPERS

- GetCDMTableId — Looks up a value in a CDM table by key and replaces the input with the matching value.
- SaveToVirtualField — Stores a single source value into a named virtual field on a CDM table.

DATE

- DayFromDate — Returns the day of the month from a date input.
- HourFromDate — Returns the hour of the day from a date input.
- MinuteFromDate — Returns the minute of the hour from a date input.
- MonthFromDate — Returns the month of the year from a date input.
- SecondFromDate — Returns the second of the minute from a date input.
- YearFromDate — Returns the year from a date input.

LOGICAL

- Not — Reverses the value of a boolean.

MAPPING-TABLE

- CSVMap — Maps the input string to a value looked up in an external CSV file by source-column key.
- Map — Maps the input using a static key/value table embedded in the rule's parameters.
- ValueMap — Resolves a vocabulary value code to its standard concept ids using the OMOP CDM vocabulary tables.

STRING

- InsertStringAt — Inserts a static string at a given position in a string input.
- PostPad — Pads a string at the end with a given character until it reaches the requested size.
- PrePad — Pads a string at the beginning with a given character until it reaches the requested size.
- Regex — Selects from the input the substring matching a regular expression.
- Split — Splits a `STRING` into an `ARRAY` of pieces on a delimiter.
- SubString — Returns a substring of the input string.

- Trim — Removes whitespace from both ends of a string.
- Truncate — Truncates a string to at most the given number of characters.

TYPE-CONVERSION

- AsDate — Transforms any type of input into a date.
- AsFloat — Transforms any type of input into a floating point value.
- AsInt — Transforms any type of input into an integer.
- AsString — Transforms any type of input into a string.

VOCABULARY

- UsagiMap — Maps the input to a concept id using a Usagi map file.
- VocabMap — Maps a vocabulary code to a standard concept id using the OMOP CDM vocabulary.
- VocabSourceId — Maps a vocabulary code to a source concept id using the OMOP CDM vocabulary.

AsDate

`mapping` category: type-conversion no `||`

Transforms any type of input into a date.

Integers are interpreted as seconds, floats are rounded and interpreted as seconds, and strings are parsed into a date using a best-effort algorithm. The string parser is not learning, which means it does not always get the dates right. To be explicit about the format, supply a `strftime`-style format template as the first parameter.

PARAMETERS

1 — `format` *Optional* — accepts: STRING

A `strftime`-style format template applied when the input is a string. Optional and only meaningful for string input.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `DATE`

EXAMPLES

Default best-effort string parser

```
AsDate()
```

Explicit format template

```
AsDate("%d%m%y-%H%M%s")
```

SEE ALSO

AsFloat, AsInt, AsString

REFERENCES

- strftime conversion specifiers

AsFloat

`mapping` category: type-conversion no `||`

Transforms any type of input into a floating point value.

Integers are simply cast to floats and dates are stored as their seconds representation. Strings are parsed as a float; if parsing fails, the record fails transformation. Supply a default value to substitute when the string cannot be parsed.

PARAMETERS

1 — `default` *Optional* — accepts: FLOAT, NULL

Default float value used when the input is a string that cannot be parsed as a float. May be `NULL`.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `FLOAT`

EXAMPLES

Default — fail on unparseable input

```
AsFloat()
```

Default to 0.0 when unparseable

```
AsFloat(0.0)
```

Default to NULL when unparseable

```
AsFloat(NULL)
```

SEE ALSO

AsInt, AsDate, AsString

AsInt

`mapping` category: type-conversion no `||`

Transforms any type of input into an integer.

Floats are rounded down. Dates are stored as their seconds representation. Strings are parsed as an integer; if parsing fails the record fails transformation. Supply a default value to substitute when the string cannot be parsed.

PARAMETERS

1 — `default` *Optional* — accepts: INT, NULL

Default integer value used when the input is a string that cannot be parsed as an integer. May be `NULL`.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `INT`

EXAMPLES

Default — fail on unparseable input

```
AsInt()
```

Default to 42 when unparseable

```
AsInt(42)
```

SEE ALSO

AsFloat, AsDate, AsString

AsString

`mapping` category: type-conversion no `||`

Transforms any type of input into a string.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

Default

```
AsString()
```

SEE ALSO

AsInt, AsFloat, AsDate

Assert

`mapping` category: assertion no `||`

Fails the record unless the input matches a value or one entry of an array of values.

Takes either a single parameter that the input must match or an array of parameters. The input must match one of the supplied values; otherwise the rule fails and the record is dropped. An optional comment is written to the log when the assertion does not match.

PARAMETERS

1 — `expected` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL, array

A value the input must match, `NULL`, or an array of values with the same type as the input. The input must equal one of these values.

2 — `comment` *Optional* — accepts: STRING

A comment written to the log when the assertion does not match.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `same-as-input`

EXAMPLES

Match a single value with comment

```
Assert("Error state", "An error state was entered")
```

Assert input is NULL

```
Assert(NULL)
```

Match any of an array of values

```
Assert([0.0, 0.1, 0.2], "Out of limits")
```

SEE ALSO

AssertNot, AssertVocabDomain

AssertNot

mapping | category: assertion | no `||`

Fails the record when the input matches a value or one entry of an array of values.

Takes either a single parameter that the input must not match or an array of parameters. If the input equals any of the supplied values the rule fails and the record is dropped. An optional comment is written to the log when the assertion does not match.

PARAMETERS

1 — `forbidden` *Optional* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL, array

A value, `NULL`, or an array of values that the input must not equal. Defaults to `NULL`.

2 — `comment` *Optional* — accepts: STRING

A comment written to the log when the assertion does not match.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `same-as-input`

EXAMPLES

Reject a single value

```
AssertNot("Error state")
```

Reject NULL with a comment

```
AssertNot(NULL, "Null value seen")
```

Reject any of an array of values

```
AssertNot([0, 1, 2])
```

SEE ALSO

Assert, AssertVocabDomain

AssertVocabDomain

`mapping` category: assertion no `||`

Fails the record unless the input concept id belongs to a named OMOP vocabulary domain.

Takes a domain name and the input concept id. The rule fails the record unless the concept id is in the named OMOP CDM vocabulary domain. Setting the second parameter to `TRUE` allows the special concept id `0`.

PARAMETERS

1 — `domain` *Mandatory* — accepts: STRING

The name of the OMOP CDM domain the input concept must belong to.

2 — `allowZero` *Mandatory* — accepts: BOOLEAN

When `TRUE`, the concept id `0` is accepted. When `FALSE`, `0` causes the assertion to fail.

3 — `comment` *Optional* — accepts: STRING

A comment written to the log when the assertion does not match.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `same-as-input`

EXAMPLES

Concept must be in the Condition domain (0 allowed)

```
AssertVocabDomain("Condition", TRUE)
```

Concept must be Procedure with custom failure message

```
AssertVocabDomain("Procedure", FALSE, "Must be Procedure")
```

SEE ALSO

Assert, AssertNot, VocabMap, VocabSourceId

At

mapping | category: array | no `||` | new-mode: required | resultSize: 1

Returns the element of an `ARRAY` at the given 0-based index as a scalar.

`At(N)` extracts a single element from an `ARRAY` value. The pipeline must carry a single `ARRAY`-typed value; the rule replaces it with a scalar copy of `array[N]`. Out-of-range indices (negative or `>= size`) return `NULL` rather than throwing.

The element type is inherited from the destination CDM field (the rule reports `UNKNOWN` from `mapType` to defer typing to the surrounding context).

Available in new mode only.

PARAMETERS

1 — `index` *Mandatory* — accepts: INT

0-based index into the array. Out-of-range indices produce `NULL`.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

Type matches the destination CDM field; element values are copied out of the array. Out-of-range indices yield `NULL`.

REQUIRES

- Exactly one ARRAY-typed pipeline input.

EXAMPLES

First element of a collected array

```
Use(:a, :b, :c)[3] => At(0)
```

Middle element (test095)

```
Use(:a, :b, :c)[3] => At(1)
```

ERRORS

- `` `At': input must be ARRAY-typed ``

  The pipeline value reaching the rule is not an `ARRAY`.

- `` `At' requires exactly one integer parameter ``

  Wrong arity or non-integer parameter (e.g. a string or field reference).

- `` `At' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

InsertAt, RemoveAt, ReplaceAt, Size, Remove

CSVMap

`mapping` category: mapping-table supports `||` resultSize: N

Maps the input string to a value looked up in an external CSV file by source-column key.

`CSVMap` looks up the input value in a CSV file registered with the engine, using `sourceColumn` as the lookup key, and returns the value from `destinationColumn` of the matching row. CSV files are loaded through the engine's `csvmap directory` configuration.

CSV columns are stored as strings, so the input must be `STRING`-typed — use `AsString()` upstream for numeric or date inputs. When the lookup fails:

- With a fourth (default) parameter, the default value is returned and `successfulMapping` is set to `false`.
- Without a default, the row is marked unsuccessful so that conditional chaining (`||`) can fall through to a fallback rule.

CSVMap supports multi-sized results: `[]` returns every matching row, `[N]` requires exactly `N`. With multiple matches, all values are emitted as additional results.

PARAMETERS

1 — `file` *Mandatory* — accepts: STRING

Name of the CSV file relative to the directory pointed to by the `csvmap directory` clause in `etl.conf`.

2 — `sourceColumn` *Mandatory* — accepts: STRING

Header name of the column to look up the input value against.

3 — `destinationColumn` *Mandatory* — accepts: STRING

Header name of the column whose value is returned on match.

4 — `default` *Optional* — accepts: STRING, NULL

Default value used when no match is found. Without this parameter, unmatched rows are marked unsuccessful so that `||`-chains can take over.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

REQUIRES

- The CSV file must be registered under the configured `csvmap directory`.
- Allows multi-sized results.

EXAMPLES

Single-result lookup with default (test064)

```
CSVMap("lookup.csv", "code", "description", "MISSING")
```

Multi-result lookup, unlimited

```
CSVMap("multi_lookup.csv", "group_code", "member_name", "MISSING")[]
```

Conditional chaining onto a Map fallback

```
CSVMap("multi_lookup.csv", "member_id", "member_name") ||
Map([["This", "is"], ["not", "used"]], "MISSING")
```

ERRORS

- `CSVMap cannot handle a null value as input`

  A `NULL` reached the rule and no default value is configured.

- `The file` X' doesn't exist in the csvmap directory`

  The CSV file is missing or not registered with the engine.

- `CSVMap requires STRING input`

  A non-`STRING` input field was used. Convert with `AsString()` first.

- `CSVMap accepts three to four parameters`

  Wrong arity or non-string parameter type.

SEE ALSO

Map, ValueMap, UsagiMap, VocabMap

DayFromDate

mapping category: date no `||`

Returns the day of the month from a date input.

Expects a date as input and returns an integer representing the day of the month (1–31). Useful for the `day_of_birth` field in the `person` table.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
DayFromDate()
```

SEE ALSO

MonthFromDate, YearFromDate, HourFromDate, MinuteFromDate, SecondFromDate, MergeDate

GetCDMTableId

`mapping` category: cdm-helpers no `||`

Looks up a value in a CDM table by key and replaces the input with the matching value.

Maps the input using two fields in a single CDM table — one acts as a key, the other as a value. If the input matches the key, it is replaced with the corresponding value. If the key cannot be found and no default is given, the record mapping fails.

A common use is to map IDs in a source table to IDs in the OMOP CDM. The best practice is to store the source ID in `<table>_source_value` and the CDM table id in `<table>_id`; these are the default key and value field names used when only the table is specified.

PARAMETERS

Default fields, optional default value

1 — `table` *Mandatory* — accepts: table

The CDM table to collect the key and value from, in `@`-notation. With this form the key field defaults to `<table_name>_source_value` and the value field to `<table_name>_id`.

2 — `default` *Optional* — accepts: any

A default value used when the key cannot be found. The type must match the value field's type.

Explicit key/value fields

1 — `table` *Mandatory* — accepts: table

The CDM table, in `@`-notation.

2 — `key` *Mandatory* — accepts: field

The key field on the CDM table, in `:`-notation.

3 — `value` *Mandatory* — accepts: field

The value field on the CDM table, in `:`-notation.

4 — `default` *Optional* — accepts: any

Default value used when the key cannot be found. The type must match the value field's type.

INPUT TYPE

The input type must match the type of the key field.

OUTPUT TYPE

The output type matches the type of the value field.

REQUIRES

- The table used must exist.
- Either both the key and value field are specified or neither is.
- Both fields must be available in the table.
- If a default value is set, its type must match that of the value.

EXAMPLES

Default fields (`<table>_source_value`, `<table>_id`)

```
GetCDMTableId(@location)
```

Explicit key and value fields

```
GetCDMTableId(@location, :location_source_value, :location_id)
```

Default value when key not found

```
GetCDMTableId(@location, 0)
```

Explicit fields with default

```
GetCDMTableId(@location, :location_source_value, :location_id, 0)
```

SEE ALSO

GetCDMFieldValue, GetCDMVisitId

HighInt

mapping | category: arithmetic | no `||`

Maps a floating point value to an integer by rounding the value up.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `FLOAT`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
HighInt()
```

SEE ALSO

AsInt, AsFloat

HourFromDate

mapping category: date no `||`

Returns the hour of the day from a date input.

Expects a date as input and returns an integer representing the hour of the day (0–23).

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
HourFromDate()
```

SEE ALSO

DayFromDate, MinuteFromDate, SecondFromDate, MonthFromDate, YearFromDate, MergeDate

InsertAt

`mapping` category: array no `||` new-mode: required resultSize: 1

Inserts a scalar value at the given 0-based index of an `ARRAY`, returning a new `ARRAY`.

`InsertAt(N, value)` inserts a scalar at position `N` of the input array; existing elements at and after `N` shift one position right. The index is clamped to `[0, size]`: negative values insert at the front, indices beyond the end append at the back.

The pipeline must carry a single `ARRAY`-typed value; the rule produces a new `ARRAY` of length `size + 1`.

Available in new mode only.

PARAMETERS

1 — `index` *Mandatory* — accepts: INT

Insertion position. Clamped to `[0, size]` — negative indices insert at the front, indices beyond the end append.

2 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL

A scalar literal to insert. `ARRAY` values and field references are not accepted.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Exactly one ARRAY-typed pipeline input.

EXAMPLES

Insert a header element at the front (test095)

```
Use(:a, :b, :c)[3] => InsertAt(0, "first")
```

Append by passing the array's length

```
Use(:a, :b)[2] => InsertAt(99, "appended")
```

ERRORS

- `` `InsertAt': input must be ARRAY-typed ``

  The pipeline value reaching the rule is not an `ARRAY`.

- `` `InsertAt' requires two parameters: an integer index and a scalar value ``

  Wrong arity, non-integer index, or non-scalar value.

- `` `InsertAt' second parameter must be a scalar value ``

  An `ARRAY` literal or unknown-type expression was used as the value parameter.

- `` `InsertAt' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

At, RemoveAt, ReplaceAt, Append, Size

InsertStringAt

`mapping` `category: string` `no ||`

Inserts a static string at a given position in a string input.

Inserts the static string into the input at the specified position. If the position is past the end of the input string, nothing is inserted.

PARAMETERS

1 — `position` *Mandatory* — accepts: INT

Zero-based offset where the static string is inserted. If past the end of the input, nothing is inserted.

2 — `text` *Mandatory* — accepts: STRING

The static string to insert.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

Insert at position 5

```
InsertStringAt(5, "Inserted text")
```

SEE ALSO

PrePad, PostPad, SubString, Concat

Map

`mapping` category: mapping-table supports `||`

Maps the input using a static key/value table embedded in the rule's parameters.

The first parameter is an array of two-entry arrays: the first entry is the key the input must match, the second is the replacement value. The rule accepts any input type; multiple key types can coexist in the map, but every entry must produce the same output type. To match string input the key may be a regular expression in `//`-notation.

Behaviour when the input matches no key:

- **With a default value** — the default is returned. `successfulMapping` is `true`. Downstream rules see the default, exactly as if it had been a successful mapping.
- **Without a default value** — the input is **passed through unchanged** and `successfulMapping` is set to `false`. Inside a `||` chain that's the cue for the next conditional rule to fire. **Outside** a `||` chain the original input value flows on. This is a footgun: a `STRING` input on a `Map` whose values are `INT` will write the literal string into a CDM integer field, which either errors out at the load boundary or inserts garbage depending on driver strictness.

The standard pattern for terminal `Map` rules (not part of a `||` chain) is therefore `Map([..., default_value)` followed by an `AssertNot(default_value)` — the default sentinel turns "no match" into a logged, skipped row rather than a corrupted insert.

PARAMETERS

1 — `map` *Mandatory* — accepts: array

An array of `[key, value]` pairs. The input is matched against each key; the first match's value is returned. Keys may be any input type and may include regular expressions for string inputs.

2 — `default` *Optional* — accepts: any

A default value used when no key matches. Its type must match the output type of the entries in the map.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

The output type matches the output type of the entries in the map.

REQUIRES

- At least one entry must exist in the mapping array.
- The inner arrays of the first parameter must each have two entries.
- All inner arrays must produce the same output type.
- A default value, if given, must match the output type of the inner arrays.

EXAMPLES

String to integer map

```
Map([["F", 8532], ["M", 8507]])
```

Integer to integer map with default

```
Map([[1, 8532], [2, 8507]], 0)
```

Regex string keys

```
Map([[/F(emale)?/, 8532], [/M(ale)?/, 8507]])
```

SEE ALSO

UsagiMap, VocabMap, VocabSourceId

MinuteFromDate

mapping category: date no `||`

Returns the minute of the hour from a date input.

Expects a date as input and returns an integer representing the minute (0–59).

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
MinuteFromDate()
```

SEE ALSO

DayFromDate, HourFromDate, SecondFromDate, MonthFromDate, YearFromDate, MergeDate

MonthFromDate

`mapping` `category: date` `no ||`

Returns the month of the year from a date input.

Expects a date as input and returns an integer representing the month of the year (1–12). Useful for the `month_of_birth` field in the `person` table.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
MonthFromDate()
```

SEE ALSO

DayFromDate, YearFromDate, HourFromDate, MinuteFromDate, SecondFromDate, MergeDate

Not

mapping category: logical no `||`

Reverses the value of a boolean.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `BOOLEAN`

OUTPUT TYPE

- `BOOLEAN`

EXAMPLES

Default

```
Not()
```

SEE ALSO

And, Or

PostPad

mapping | category: string | no `||`

Pads a string at the end with a given character until it reaches the requested size.

Expands the input string to the requested size by appending the padding character. If the input string is already at or above that size, it is returned unchanged.

PARAMETERS

1 — `size` *Mandatory* — accepts: INT

The size the input string should have after padding.

2 — `padChar` *Optional* — accepts: STRING

The character used to pad the string. Defaults to a single space.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

REQUIRES

- The padding character can only be a single character and not a string.

EXAMPLES

Right-pad to size 5 with zeros

```
PostPad(5, "0")
```

Right-pad to size 10 with spaces

```
PostPad(10)
```

SEE ALSO

PrePad, Truncate, Trim, InsertStringAt

PrePad

mapping category: string no `||`

Pads a string at the beginning with a given character until it reaches the requested size.

Expands the input string to the requested size by prepending the padding character. If the input string is already at or above that size, it is returned unchanged.

PARAMETERS

1 — `size` *Mandatory* — accepts: INT

The size the input string should have after padding.

2 — `padChar` *Optional* — accepts: STRING

The character used to pad the string. Defaults to a single space.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

REQUIRES

- The padding character can only be a single character and not a string.

EXAMPLES

Left-pad to size 5 with zeros

```
PrePad(5, "0")
```

Left-pad to size 10 with spaces

```
PrePad(10)
```

SEE ALSO

PostPad, Truncate, Trim, InsertStringAt

Regex

`mapping` category: string no `||`

Selects from the input the substring matching a regular expression.

Selects the text from the input that matches the supplied regular expression. If the input does not match, an empty string is returned.

PARAMETERS

1 — `pattern` *Mandatory* — accepts: regex

The regular expression to match. Use `/.../` notation to delimit the pattern.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

Capture a phone number prefix

```
Regex(/\(01\)[0-9]+/)
```

Match anything

```
Regex(//)
```

SEE ALSO

SubString, Map

Remove

`mapping` `category: array` `no `||`` `new-mode: required` `resultSize: 1`

Drops every element of an `ARRAY` equal to the given scalar value.

`Remove(value)` filters an `ARRAY`, dropping every element that equals `value`. The result is a new `ARRAY` containing the surviving elements in their original order. `NULL` elements are dropped unconditionally.

Typical use is stripping placeholder values from a collected `ARRAY`, for example after `Use(...)[N]` produced fields containing `"."` for missing values.

Available in new mode only.

PARAMETERS

1 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL

The scalar value to drop. Field references and `ARRAY` literals are not accepted. Equality semantics follow the per-type `Value` operator.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Exactly one ARRAY-typed pipeline input.

EXAMPLES

Strip placeholder strings from a collected ARRAY (test083)

```
Use(:int1, :int2, :int3, :int4)[4] => Remove(".") | AsInt()
```

Drop a sentinel integer

```
:codes => Remove(0)
```

ERRORS

- `` `Remove': input must be ARRAY-typed ``

  The pipeline value reaching the rule is not an `ARRAY`.

- `` `Remove' requires exactly one scalar value parameter ``

  Wrong arity, ARRAY literal, or field reference used as the parameter.

- `` `Remove' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

RemoveAt, Without, Unique, Size, At

RemoveAt

`mapping` category: array no `||` new-mode: required resultSize: 1

Removes the element at the given 0-based index from an `ARRAY`.

`RemoveAt(N)` returns a new `ARRAY` with the element at index `N` dropped. Out-of-range indices are silently ignored — the array is returned unchanged.

The index parameter accepts either a literal integer or a field reference. Field-reference indices are resolved per row from the pipeline values; the referenced field must hold an `INT`. The transformer also supports infix arithmetic in the index parameter (e.g. `:len - 1`), which is desugared into a preceding `Subtract` step.

Available in new mode only.

PARAMETERS

1 — `index` *Mandatory* — accepts: INT, field

0-based index, either as an integer literal or a field reference to an `INT`-valued pipeline field.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Exactly one ARRAY-typed pipeline input.
- When index is a field reference the referenced field must hold an INT.

EXAMPLES

Remove a fixed position

```
:array => RemoveAt(0)
```

Remove a position computed at runtime

```
:array => RemoveAt(:idx)
```

Remove the last element via infix arithmetic

```
:array => RemoveAt(:len - 1)
```

ERRORS

- `` `RemoveAt': input must be ARRAY-typed ``

  The pipeline value reaching the rule is not an `ARRAY`.

- `` `RemoveAt': index field `:idx' must be an integer` ``

  The referenced field is not an `INT`.

- `` `RemoveAt': index field :idx' not found in pipeline` ``

  The referenced field is not present in the current pipeline state.

- `` `RemoveAt' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

At, InsertAt, ReplaceAt, Remove, Size

ReplaceAt

`mapping` category: array no `||` new-mode: required resultSize: 1

Replaces the element at the given 0-based index of an `ARRAY` with a new value.

`ReplaceAt(N, value)` returns a new `ARRAY` with the element at index `N` replaced by `value`. Out-of-range indices are silently ignored — the array is returned unchanged.

Both the index and the replacement value can be literals or field references; field references are resolved per row from the current pipeline state. Field-reference indices must be `INT`-valued, while field-reference values can be any scalar. The transformer also supports infix arithmetic in the index parameter (e.g. `:len - 1`).

Available in new mode only.

PARAMETERS

1 — `index` *Mandatory* — accepts: INT, field

0-based index, either as an integer literal or a field reference to an `INT`-valued pipeline field.

2 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL, field

Replacement value. May be a scalar literal or a field reference to any non-`ARRAY` pipeline field.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Exactly one ARRAY-typed pipeline input.
- When index is a field reference the referenced field must hold an INT.

EXAMPLES

Replace a literal position with a literal value

```
:array => ReplaceAt(2, "new")
```

Use field-reference index and value

```
:array => ReplaceAt(:idx, :val)
```

Replace the last element via infix arithmetic

```
:array => ReplaceAt(:len - 1, :val)
```

ERRORS

- `` `ReplaceAt': input must be ARRAY-typed ``

The pipeline value reaching the rule is not an `ARRAY`.

- `` `ReplaceAt': index field `` :idx' must be an integer`

  The referenced index field is not an `INT`.

- `` `ReplaceAt': value field `` :val' could not be resolved`

  The referenced value field is not in the pipeline or has no value.

- `` `ReplaceAt' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

At, InsertAt, RemoveAt, Remove, Size

SaveToVirtualField

`mapping` category: cdm-helpers no `||`

Stores a single source value into a named virtual field on a CDM table.

Special mapping rule used to store a single value from a source table into a virtual field on a CDM table. Virtual fields are not visually represented in Rabbit in a Hat, so this rule is placed on a transaction to an otherwise unrelated field. The transaction effectively writes to the named virtual field instead.

PARAMETERS

1 — `virtualField` *Mandatory* — accepts: field

The name of the virtual field, in `:`-notation.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

Does not produce output to a CDM field; the value is stored only in the named virtual field.

REQUIRES

- Can only save to virtual fields. For actual fields make a transition in Rabbit in a Hat.
- The virtual field referenced must be defined on the CDM table.

EXAMPLES

Save to a virtual field

```
SaveToVirtualField(:care_site_map)
```

SEE ALSO

GetCDMFieldValue, GetCDMTableId

SecondFromDate

mapping category: date no `||`

Returns the second of the minute from a date input.

Expects a date as input and returns an integer representing the second (0–59).

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
SecondFromDate()
```

SEE ALSO

DayFromDate, HourFromDate, MinuteFromDate, MonthFromDate, YearFromDate, MergeDate

Split

mapping | category: string | no `||` | new-mode: required | resultSize: 1

Splits a `STRING` into an `ARRAY` of pieces on a delimiter.

`Split(delimiter)` divides the input `STRING` into pieces wherever the (possibly multi-character) `delimiter` occurs, producing an `ARRAY` of `STRING` values. Splitting follows the usual explicit-separator semantics: adjacent delimiters and leading/trailing delimiters yield empty pieces, and a delimiter that does not occur yields a single-element `ARRAY` containing the whole input.

Typically used to fan a compound source value into its parts for further processing — for example a `"low-high"` reference range into its two bounds, or a space-separated code into tokens — then chained into `| Rule()` (per-element), `At(n)`, `Size()`, or a `LoopOver` expansion.

`Split` returns the whole `ARRAY` itself, so it does **not** take the `[]` suffix — write `:value => Split("-")`, the same way as `Remove` and `Without`. (The `[]` suffix is for scalar-per-result collectors such as `UsagiMap()[]` or `Use(...)[N]`; using it here would also drop the STRING element-type hint that a following `| Rule()` needs.)

Available in new mode only.

PARAMETERS

1 — `delimiter` *Mandatory* — accepts: STRING

The separator to split on. Must be a non-empty string literal; field references and `ARRAY` literals are not accepted.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Exactly one STRING-typed pipeline input.

EXAMPLES

Split a reference range into its two bounds, parsing each

```
:range => Split("-") | AsFloat()
```

Tokenise a space-separated value

```
:blood_group => Split(" ")
```

Split then take the first piece

```
:code => Split(".") => At(0)
```

ERRORS

- `` `Split' expects varchar but got `` `

The pipeline value reaching the rule is not a `STRING`.

- `` `Split' requires exactly one non-empty string delimiter ``

Wrong arity, an empty delimiter, an `ARRAY` literal, or a field reference.

- `Split' is only available in new mode`

  New mode is not enabled in `etl.conf`.

SEE ALSO

SubString, Regex, Remove, At, Size

SubString

`mapping` `category: string` `no ||`

Returns a substring of the input string.

Returns the substring beginning at the given position with the given length. If the requested range extends past the end of the input, the available characters are returned.

PARAMETERS

1 — `position` *Mandatory* — accepts: INT

Zero-based index of the first character of the substring.

2 — `length` *Optional* — accepts: INT

Length of the substring. If omitted, the substring runs from the starting position to the end of the input.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

5 characters starting at index 2

```
SubString(2, 5)
```

From index 10 to end of string

```
SubString(10)
```

SEE ALSO

Regex, Trim, Truncate

Trim

mapping category: string no `||`

Removes whitespace from both ends of a string.

Strips leading and trailing whitespace from the input string.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

Default

```
Trim()
```

SEE ALSO

Truncate, PrePad, PostPad, SubString

Truncate

mapping | category: string | no `||`

Truncates a string to at most the given number of characters.

If the input string is longer than the configured maximum length, it is truncated; if it is shorter, it is returned unchanged.

PARAMETERS

1 — `maxLength` *Mandatory* — accepts: INT

Maximum length the input string can have after truncation.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

EXAMPLES

Truncate to 5 characters

```
Truncate(5)
```

Truncate to 40 characters

```
Truncate(40)
```

SEE ALSO

Trim, SubString, PrePad, PostPad

UsagiMap

`mapping` `category: vocabulary` `supports ||` `resultSize: N`

Maps the input to a concept id using a Usagi map file.

Looks up the input in the named Usagi file and returns the matching concept id(s). If the same key appears multiple times in the Usagi map, all matching output values can be emitted as a multi-sized result.

PARAMETERS

1 — `file` *Mandatory* — accepts: STRING

The Usagi file name relative to the directory pointed to by the `usagi directory` clause in the ETL-Engine configuration file.

2 — `default` *Optional* — accepts: any

Default value used when no match is found in the Usagi file.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `INT[]`

REQUIRES

- Allows multi-sized results.

EXAMPLES

Single-result mapping

```
UsagiMap("usagi-file.csv")
```

With default value

```
UsagiMap("map.csv", 0)
```

Multi-sized result, unlimited

```
UsagiMap("multi-sized-map.csv")[]
```

Multi-sized result, limited to 5

```
UsagiMap("multi-sized-limited-map.csv")[5]
```

SEE ALSO

VocabMap, VocabSourceId, Map

ValueMap

mapping | category: mapping-table | supports `||` | resultSize: N

Resolves a vocabulary value code to its standard concept ids using the OMOP CDM vocabulary tables.

`ValueMap` is the value-domain analog of `VocabMap`. Instead of mapping a source concept code to its standard concept id, it maps a value code (e.g. a measurement unit, a gender code, a drug strength code) using the OMOP `concept_relationship` `Maps to value` linkage. The rule returns every matching standard concept id, supporting multi-sized results.

Typical use: resolving a free-text or coded measurement value to its standard concept id when populating `*_value_as_concept_id` fields.

PARAMETERS

1 — `vocabulary` *Mandatory* — accepts: STRING

The vocabulary used in the mapping (e.g. `"ICD10CM"`, `"LOINC"`).

2 — `default` *Optional* — accepts: INT, NULL

Default value used when no match is found.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `INT`

REQUIRES

- Allows multi-sized results.

EXAMPLES

Multi-sized ICD10CM value mapping (test063)

```
ValueMap("ICD10CM")[]
```

Single-result with default

```
ValueMap("LOINC", 0)
```

SEE ALSO

VocabMap, VocabSourceId, UsagiMap, Map, CSVMap

VocabMap

mapping | category: vocabulary | supports `||`

Maps a vocabulary code to a standard concept id using the OMOP CDM vocabulary.

Resolves the input concept code to its mapped standard concept id by querying the OMOP CDM vocabulary:

```
select concept_id_1
  from concept
     inner join concept_relationship
        on concept_id = concept_id_1
  where relationship_id = 'Maps to'
     and vocabulary_id = :vocabulary_id
     and concept_code = :concept_code
```

PARAMETERS

1 — `vocabulary` *Mandatory* — accepts: STRING

The vocabulary used in the mapping.

2 — `default` *Optional* — accepts: INT, NULL

Default value used when no match is found in the vocabulary.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `INT`

EXAMPLES

SNOMED with default NULL

```
VocabMap("SNOMED")
```

ATC with explicit default 0

```
VocabMap("ATC", 0)
```

SEE ALSO

VocabSourceId, UsagiMap, AssertVocabDomain

VocabSourceId

mapping | category: vocabulary | supports `||`

Maps a vocabulary code to a source concept id using the OMOP CDM vocabulary.

Resolves the input concept code to its source concept id by querying the OMOP CDM vocabulary:

```
select concept_id_2
  from concept
     inner join concept_relationship
        on concept_id = concept_id_1
  where relationship_id = 'Maps to'
     and vocabulary_id = :vocabulary_id
     and concept_code = :concept_code
```

PARAMETERS

1 — `vocabulary` *Mandatory* — accepts: STRING

The vocabulary used in the mapping.

2 — `default` *Optional* — accepts: INT, NULL

Default value used when no match is found in the vocabulary.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `INT`

EXAMPLES

SNOMED with default NULL

```
VocabSourceId("SNOMED")
```

ATC with explicit default 0

```
VocabSourceId("ATC", 0)
```

SEE ALSO

VocabMap, UsagiMap, AssertVocabDomain

YearFromDate

mapping | category: date | no `||`

Returns the year from a date input.

Expects a date as input and returns an integer representing the year. Useful for the `year_of_birth` field in the `person` table.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `DATE`

OUTPUT TYPE

- `INT`

EXAMPLES

Default

```
YearFromDate()
```

SEE ALSO

DayFromDate, MonthFromDate, HourFromDate, MinuteFromDate, SecondFromDate, MergeDate

5.1.3 Aggregating Rules

Aggregating rules

Aggregating rules combine many values into one.

AGGREGATION

- Avg — Returns the mean of the given parameters or inputs, folding ARRAY inputs element-by-element.
- Count — Returns the number of non-NULL values among the given parameters or inputs, counting ARRAY elements individually.
- IgnoreValue — Removes a single input value from the rule logic context.
- Max — Returns the largest value among the given parameters or inputs, folding ARRAY inputs element-by-element.
- Min — Returns the smallest value among the given parameters or inputs, folding ARRAY inputs element-by-element.
- Selector — Picks one of two values based on a boolean or a comparison between two values.
- SetValue — Emits the supplied parameter value as the output.
- Sum — Returns the sum of the given parameters or the available inputs, folding ARRAY inputs element-by-element.
- Unique — Deduplicates values from one or more inputs; in new mode also flattens `ARRAY` inputs into one `ARRAY`.
- Use — Selects which field or local variable becomes the output of the rule chain.

ARITHMETIC

- Add — Adds all values given to the rule.
- AsDays — Converts an integer number of seconds into the equivalent number of days.
- AsHours — Converts an integer number of seconds into the equivalent number of hours.
- AsMinutes — Converts an integer number of seconds into the equivalent number of minutes.
- AsMonths — Converts an integer number of seconds into the equivalent number of months.
- AsSeconds — Returns the input value (in seconds) unchanged — completes the time-unit set.
- AsYears — Converts an integer number of seconds into the equivalent number of years.
- Divide — Divides a dividend by all subsequent values or all available inputs.
- Multiply — Multiplies all values given to the rule, or all available inputs.
- Subtract — Subtracts the remaining values from the first, with date- and float-aware promotion.

ARRAY

- Append — Concatenates two or more `ARRAY` inputs into a single `ARRAY`, preserving order.
- CartesianOf — Cartesian product of N input `ARRAY` values, returning an `ARRAY` of nested-`ARRAY` combinations.
- Fill — Creates an `ARRAY` of `count` copies of `value`.
- Size — Returns the number of elements in an `ARRAY` value as an `INT`.
- Spawn — Expands an `ARRAY` value into individual rows, fanning the chain out to one row per element.
- Tuple — Packs two or more named fields into a single `TUPLE` value (or `ARRAY<TUPLE>` when any input is an `ARRAY`).
- TupleGet — Extracts the element at the given 0-based index from a `TUPLE`-valued field.
- Without — Set difference of two `ARRAY` inputs — elements of `:source` not present in `:exclude`.

CDM-HELPERS

- AutoGenId — Generates a sequence of unique identifiers, optionally sized by another multi-sized field.
- GetCDMFieldValue — Returns the value of another field on the same CDM table.
- GetCDMValue — Looks up a field's value on another CDM table by an exact (optionally composite) key.
- GetCDMVisitId — Looks up a visit id on a named CDM table by key with the narrowest matching date interval.

- GetEnvironmentValue — Reads a value from the engine's environment database and returns it as a `STRING`.
- NumberOfResults — Returns the number of results produced by a sibling field, or by the current execution context.

COMPARISON

- Equal — Returns TRUE when all values (parameters or inputs) are equal.
- GreaterThan — Returns TRUE when each parameter is strictly greater than the next.
- GreaterThanOrEqual — Returns TRUE when each parameter is greater than or equal to the next.
- LessThan — Returns TRUE when each parameter is strictly less than the next.
- LessThanOrEqual — Returns TRUE when each parameter is less than or equal to the next.
- NotEqual — Returns TRUE when all values (parameters or inputs) are different.

DATE

- Days — Converts a number of days into the corresponding number of seconds.
- Hours — Converts a number of hours into the corresponding number of seconds.
- MergeDate — Combines several integer or date components into a single date.
- Minutes — Converts a number of minutes into the corresponding number of seconds.
- Months — Converts a number of months into the corresponding number of seconds (30-day months).
- Seconds — Returns the input number of seconds unchanged — completes the time-unit set.
- Years — Converts a number of years into the corresponding number of seconds (365-day years).

LOGICAL

- And — Boolean AND across all parameters or all available inputs.
- Or — Boolean OR across all parameters or all available inputs.

STRING

- Concat — Concatenates any number of strings, mixing field references and static literals.

Add

aggregating | category: arithmetic | no `||` | resultSize: 1

Adds all values given to the rule.

If no parameters are provided, all available inputs are added. Each parameter can be either a field reference or a static numeric value. The output type is promoted from the inputs: `INT` if all inputs are integers, `FLOAT` if any input is a float, and `DATE` if any input is a date. Date inputs are added as seconds, so `Add(:date, 86400)` advances `:date` by one day.

PARAMETERS

N — `addends` *Optional* — accepts: field, INT, FLOAT

The Nth addend. May be a field reference (`:f`) or a static numeric literal. If no parameters are given, all available inputs are summed.

INPUT TYPE

- `INT`
- `FLOAT`
- `DATE`

OUTPUT TYPE

- `matches-input-promotion`

Returns `DATE` if any input is a date, `FLOAT` if any input is a float, otherwise `INT`.

REQUIRES

- Only one date can be given in the input.
- A date cannot be added to a float.
- If a float is in the input the return type will always be a float.
- If a date is in the input, the return type will always be a date.

EXAMPLES

Sum all available inputs

```
Add()
```

Static values

```
Add(1, 2)
```

Mix of fields and constants (promotes to FLOAT)

```
Add(:integer1, 0.6)
```

Date arithmetic — advance a DATE by an INT number of seconds

```
Add(:date1, :integer2, 3600)
```

ERRORS

- `An unsupported type was being added to an integer or date addition`

  A `STRING` or `BOOLEAN` value reached the rule. Only `INT`, `FLOAT`, and `DATE` are accepted.

- `Add cannot add multiple dates`

  Two or more `DATE` inputs were supplied. Only one date is permitted per `Add`.

- `Add cannot add a date and a float`

  A `DATE` input was combined with a `FLOAT`. Cast the float to int first.

SEE ALSO

Subtract, Multiply, Divide, Days, Hours, Minutes, Seconds

And

aggregating | category: logical | no `||`

Boolean AND across all parameters or all available inputs.

ANDs together two or more boolean values. The values can be supplied as parameters; if no parameters are provided, all available inputs are ANDed.

PARAMETERS

N — `operands` *Optional* — accepts: field, BOOLEAN

The Nth boolean to AND. May be a field reference (`:f`) or a static boolean. If no parameters are given, every available input is ANDed.

INPUT TYPE

- `BOOLEAN`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- At least two inputs are available.

EXAMPLES

AND all available inputs

```
And()
```

AND of static booleans

```
And(TRUE, FALSE)
```

AND of field references

```
And(:BooleanValue1, :BooleanValue2)
```

SEE ALSO

Or, Not

Append

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Concatenates two or more `ARRAY` inputs into a single `ARRAY`, preserving order.

`Append` is an array-aware aggregating rule: it accepts two or more `ARRAY`-typed inputs and returns one `ARRAY` containing every element in input order (input one's elements first, then input two's, ...).

The rule has two equivalent shapes:

- **Parameter form** — `Append(:arr1, :arr2, ...)` resolves the inputs by field name. At least two field references are required.
- **Pipeline form** — `:arr1, :arr2 => Append()` consumes all positional pipeline inputs in order.

Available in new mode only.

PARAMETERS

Parameter form

N — `arr` *Mandatory* — accepts: field

Two or more field references, each referring to an `ARRAY`-typed input field.

Pipeline form (no parameters)

No parameters accepted.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- All inputs must be ARRAY-typed.
- At least two inputs are required.

EXAMPLES

Pipeline form

```
:base, :extra => Append() => :combined
```

Parameter form

```
Append(:base, :extra)
```

Construct an ARRAY by appending a virtual extension (test094)

```
:n => AsInt() => :n_int
Fill(9, :n_int)[] => :base
SetValue([0]) => :zero
Append(:base, :zero)[]
```

ERRORS

- `` `Append': all inputs must be ARRAY-typed ``

  A non-ARRAY value reached the rule at runtime.

- `` `Append' parameter form requires at least two field references ``

  Fewer than two parameters were supplied.

- `` `Append': field `` :f' not found in input set`

  A named field reference does not match any input field.

- `` `Append' is only available in new mode (set `` new mode : "true"' in etl.conf)`

  New mode is not enabled in `etl.conf`.

SEE ALSO

Fill, CartesianOf, Unique, Without, Use

AsDays

aggregating | category: arithmetic | no `||`

Converts an integer number of seconds into the equivalent number of days.

Helper that divides the input or parameter (in seconds) by `60 * 60 * 24` to produce the equivalent number of days.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds to convert to days. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsDays()
```

From static parameter

```
AsDays(3628800)
```

SEE ALSO

AsHours, AsMinutes, AsMonths, AsSeconds, AsYears, Days

AsHours

aggregating | category: arithmetic | no `||`

Converts an integer number of seconds into the equivalent number of hours.

Helper that divides the input or parameter (in seconds) by `60 * 60` to produce the equivalent number of hours.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds to convert to hours. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsHours()
```

From static parameter

```
AsHours(10800)
```

SEE ALSO

AsDays, AsMinutes, AsMonths, AsSeconds, AsYears, Hours

AsMinutes

aggregating | category: arithmetic | no `||`

Converts an integer number of seconds into the equivalent number of minutes.

Helper that divides the input or parameter (in seconds) by `60` to produce the equivalent number of minutes.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds to convert to minutes. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsMinutes()
```

From static parameter

```
AsMinutes(240)
```

SEE ALSO

AsDays, AsHours, AsMonths, AsSeconds, AsYears, Minutes

AsMonths

aggregating | category: arithmetic | no `||`

Converts an integer number of seconds into the equivalent number of months.

Helper that divides the input or parameter (in seconds) by `60 * 60 * 24 * 30` to produce the equivalent number of months.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds to convert to months. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsMonths()
```

From static parameter

```
AsMonths(38880000)
```

SEE ALSO

AsDays, AsHours, AsMinutes, AsSeconds, AsYears, Months

AsSeconds

aggregating category: arithmetic no `||`

Returns the input value (in seconds) unchanged — completes the time-unit set.

Helper companion to `AsMinutes`, `AsHours`, `AsDays`, `AsMonths`, and `AsYears`. Since the input integer value is already in seconds, the command simply passes the value through unchanged. It exists for symmetry with the other time-unit helpers.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsSeconds()
```

From static parameter

```
AsSeconds(53)
```

SEE ALSO

AsDays, AsHours, AsMinutes, AsMonths, AsYears, Seconds

AsYears

aggregating | category: arithmetic | no `||`

Converts an integer number of seconds into the equivalent number of years.

Helper that divides the input or parameter (in seconds) by `60 * 60 * 24 * 365` to produce the equivalent number of years.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds to convert to years. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
AsYears()
```

From static parameter

```
AsYears(94608000)
```

SEE ALSO

AsDays, AsHours, AsMinutes, AsMonths, AsSeconds, Years

AutoGenId

aggregating | category: cdm-helpers | no `||` | resultSize: N

Generates a sequence of unique identifiers, optionally sized by another multi-sized field.

Creates an ID sequence. Without parameters, the sequence is unique to the field the rule is placed on. With a string parameter, the sequence is named and global — usable across tables and storable across runs via the `StoreSequence` table rule.

`AutoGenId` works with multi-sized results and can be forced to produce a fixed number of results by specifying a limit. To match an unlimited multi-sized field elsewhere, supply that field as the first parameter so `AutoGenId` produces the same number of results.

PARAMETERS

1 — `source` *Optional* — accepts: field

A field whose multi-sized result size determines how many ids to generate. Required only when matching an unlimited multi-sized result.

2 — `sequenceName` *Optional* — accepts: STRING

The name of a global sequence. Named sequences are shared across tables and can be persisted between runs using `StoreSequence`.

INPUT TYPE

Does not accept input.

OUTPUT TYPE

- `INT[]`

REQUIRES

- Allows multi-sized results.

EXAMPLES

Per-field local sequence

```
AutoGenId()
```

Per-field local sequence, exactly 2 ids

```
AutoGenId()[2]
```

Sized by another multi-sized field

```
AutoGenId(:another_field)[]
```

Named global sequence

```
AutoGenId("My Sequence")
```

Named global sequence, exactly 5 ids

```
AutoGenId("My Sequence")[5]
```

Named global sequence sized by another field

```
AutoGenId(:some_field, "My Sequence")[]
```

SEE ALSO

GetCDMTableId, GetCDMFieldValue

Avg

aggregating | category: aggregation | no `||`

Returns the mean of the given parameters or inputs, folding ARRAY inputs element-by-element.

Without parameters, `Avg` averages every available input. When parameters are given, it averages only those parameters/fields. An ARRAY input is folded element-by-element (the same way `Min` / `Max` / `Sum` / `Count` fold arrays), so `GetCDMTableId(@idx, :key)[] => Avg()` returns the mean of all matched values.

NULL values and NULL ARRAY elements are skipped — the mean is `sum(present values) / count(present values)`, mirroring SQL `AVG(expr)`. The result is always `FLOAT`. When there are no present (non-NULL) values the result is NULL (no division by zero).

`Avg` accepts only numeric (`INT` / `FLOAT`) values; dates and strings are rejected at setup.

PARAMETERS

N — `values` *Optional* — accepts: field, FLOAT, INT

The Nth value to average. May be a field reference (scalar or ARRAY) or a static number.

INPUT TYPE

- `FLOAT`
- `INT`
- `ARRAY`

OUTPUT TYPE

- `FLOAT`

EXAMPLES

Average across all inputs

```
Avg()
```

Mean of a collected ARRAY

```
GetCDMTableId(@line_index, :claim_id, :amount)[] => Avg()
```

SEE ALSO

Sum, Count, Min, Max

CartesianOf

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Cartesian product of N input `ARRAY` values, returning an `ARRAY` of nested-`ARRAY` combinations.

`CartesianOf` consumes every `ARRAY`-typed pipeline input and produces a single output `ARRAY`. Each output element is itself an `ARRAY` holding one element drawn from each input array. The combinations are emitted outer-array-first — the first input is the slowest-changing index.

With one input, `CartesianOf` returns a deep copy of the input array (identity case). With two or more inputs, the output length is the product of the input lengths. The rule is the new-mode replacement for the legacy `Cartesian` table rule.

Available in new mode only.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

An `ARRAY` whose elements are nested `ARRAY` values, one per combination.

REQUIRES

- All pipeline inputs must be ARRAY-typed.
- At least one pipeline input is required.

EXAMPLES

Two-input Cartesian product

```
:colors, :sizes => CartesianOf() => :combos
```

Three-input Cartesian product producing nested-ARRAY tuples

```
:a, :b, :c => CartesianOf() => :combos
```

ERRORS

- `` `CartesianOf': all inputs must be ARRAY-typed ``

  A non-ARRAY value reached the rule at runtime.

- `` `CartesianOf' requires at least one ARRAY input ``

  No pipeline inputs were supplied.

- `` `CartesianOf' does not accept parameters ``

  Parameters were supplied — `CartesianOf` always reads its inputs from the pipeline.

- `` `CartesianOf' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

Append, Unique, Without, Use, Spawn

Concat

aggregating | category: string | no `||`

Concatenates any number of strings, mixing field references and static literals.

Concatenates string parameters in the order they are given. Inputs are passed by `:`-notation reference; static strings can be interleaved.

PARAMETERS

N — `parts` *Mandatory* — accepts: field, STRING

The Nth component of the resulting string. May be a field reference (`:f`) or a static string. Any number of parts can be given.

INPUT TYPE

- `STRING`

OUTPUT TYPE

- `STRING`

REQUIRES

- Concat works both with and without inputs. Without inputs it can only contain static strings.

EXAMPLES

Mix fields and a separator

```
Concat(:field1, " - ", :field2)
```

Static-only — equivalent to SetValue

```
Concat("I work like SetValue")
```

Prefix label and a field

```
Concat("Field7 contains: ", :field7)
```

Three field references

```
Concat(:field1, :field2, :field3)
```

SEE ALSO

SetValue, InsertStringAt, PrePad, PostPad

Count

aggregating | category: aggregation | no `||`

Returns the number of non-NULL values among the given parameters or inputs, counting ARRAY elements individually.

Without parameters, `Count` counts every available input. When parameters are given, it counts only those parameters/fields. An ARRAY input is counted element-by-element (the same way `Min` / `Max` / `Sum` fold arrays), so `GetCDMTableId(@idx, :key)[] => Count()` returns how many rows matched.

NULL values and NULL ARRAY elements are NOT counted — `Count` is the number of present (non-NULL) values, mirroring SQL `COUNT(expr)`. The result is always `INT`. Any value type is accepted (it is only tested for NULL, never compared).

PARAMETERS

N — `items` *Optional* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

The Nth value to count. May be a field reference (scalar or ARRAY) or a static value.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`
- `ARRAY`

OUTPUT TYPE

- `INT`

EXAMPLES

Count all inputs

```
Count()
```

Count a collected ARRAY

```
GetCDMTableId(@line_index, :claim_id, :amount)[] => Count()
```

SEE ALSO

Sum, Size, Min, Max

Days

aggregating | category: date | no `||`

Converts a number of days into the corresponding number of seconds.

Helper that multiplies the input or parameter (in days) into seconds. Useful for date arithmetic where the engine stores dates as seconds.

PARAMETERS

1 — `days` *Optional* — accepts: field, INT

Number of days to convert to seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
Days()
```

From static parameter

```
Days(42)
```

SEE ALSO

Hours, Minutes, Seconds, Months, Years, AsDays

Divide

aggregating | category: arithmetic | no `||`

Divides a dividend by all subsequent values or all available inputs.

Divides the first parameter (the dividend) by all remaining parameters in order. With no additional parameters, the dividend is divided by all available inputs.

PARAMETERS

1 — `dividend` *Mandatory* — accepts: field, INT, FLOAT

The dividend of the division. May be a field reference or a static value.

N — `divisors` *Optional* — accepts: field, INT, FLOAT

The Nth divisor that the dividend is divided by. May be a field reference or a static value.

INPUT TYPE

- `FLOAT`
- `INT`

OUTPUT TYPE

- `FLOAT`

EXAMPLES

Divide the dividend by all available inputs

```
Divide(:dividend)
```

Divide the dividend by two specific divisors

```
Divide(:dividend, :divisor1, :divisor2)
```

Divide a field by a static value

```
Divide(:dividend, 10)
```

Mixed static and field divisors

```
Divide(:dividend, 3.14, :divisor2)
```

SEE ALSO

Multiply, Add, Subtract

Equal

aggregating | category: comparison | no `||`

Returns TRUE when all values (parameters or inputs) are equal.

Without parameters, the rule checks that every input is equal. With parameters, the listed values — static or field references — must all be equal to produce a `TRUE` result.

PARAMETERS

N — `values` *Optional* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

The Nth value to compare. May be a static value or a field reference. If no parameters are given, all available inputs are compared.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.

EXAMPLES

Compare all available inputs

```
Equal()
```

Compare three field references

```
Equal(:float1, :float2, :float3)
```

Mix static value and field reference

```
Equal(42, :the_answer)
```

SEE ALSO

NotEqual, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual

Fill

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Creates an `ARRAY` of `count` copies of `value`.

`Fill(value, count)` produces a fresh `ARRAY` with `count` identical elements, each a copy of `value`. Both parameters may be scalar literals or field references to scalar pipeline values; `count` must resolve to an integer. A non-positive `count` yields an empty `ARRAY`.

Typical use is constructing a base array of a known length that is then combined with other arrays via `Append`, `ReplaceAt`, or similar array-aware rules.

Available in new mode only.

PARAMETERS

1 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL, field

The element value. Each output element is an independent copy of this value.

2 — `count` *Mandatory* — accepts: INT, field

Number of elements to produce. Negative values are clamped to zero, yielding an empty `ARRAY`.

INPUT TYPE

OUTPUT TYPE

- `ARRAY`

REQUIRES

- `count` must resolve to an integer at runtime.

EXAMPLES

Build a base ARRAY then append a sentinel (test094)

```
:n => AsInt() => :n_int
Fill(9, :n_int)[] => :base
SetValue([0]) => :zero
Append(:base, :zero)[]
```

Pre-populate strings of a fixed length

```
Fill("SRC", :n_int)[] => :base
```

ERRORS

- `` `Fill': count must be an integer ``

  The `count` parameter resolved to a non-integer value at runtime.

- `` `Fill': second parameter (count) must be an integer ``

  Parse-time type check rejected a non-integer count parameter.

- `` `Fill' requires exactly two parameters ``

  Wrong arity.

- `` `Fill' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

Append, ReplaceAt, SetValue, Use, Size

GetCDMFieldValue

aggregating | category: cdm-helpers | no `||`

Returns the value of another field on the same CDM table.

Loads the value of the named field from the same CDM table as this rule. Establishes a dependency, forcing the referenced field to be calculated first. If multiple `GetCDMFieldValue` rules across the table create a circular reference, an error is thrown.

PARAMETERS

1 — `field` *Mandatory* — accepts: field

The field on the same CDM table to load the value from, in `:`-notation.

INPUT TYPE

Field reference (no value input).

OUTPUT TYPE

The output type matches the referenced field's type.

REQUIRES

- No circular reference.
- Field must be on the same table as the rule.
- Only valid in field logic, not on a mapping rule.

EXAMPLES

Read the person_id field of the same CDM table

```
GetCDMFieldValue(:person_id)
```

SEE ALSO

GetCDMTableId, GetCDMVisitId

GetCDMValue

aggregating | category: cdm-helpers | no `||`

Looks up a field's value on another CDM table by an exact (optionally composite) key.

Fetches a field's value from a named CDM table by matching one or more key fields against values from the current row — the general, exact-match companion to `GetCDMVisitId` (which matches by date interval) and `GetCDMTableId` (which returns the id via a stored index map).

This is the supported way to pull a value out of another CDM table. The rule records a build-order dependency on the looked-up table at configure time, and at run time it does a `getFromCache` lookup and returns the requested field — so the looked-up table is fully built first and the value is simply read back. It is a standalone lookup, not a join.

Keys are given as two parallel arrays, the same way the rest of the system takes key/value lists: a list of key fields on the looked-up table and a list of values (field references or literals) from the current row, matched position-by-position. The arrays must be the same length. The order you list the key fields in does not matter — the lookup is matched against the table's index by field, so it resolves the same regardless of declared key order.

If the key matches several rows, each match is a result — so on a non-unique key the rule produces a multi-sized result (use `[]` / `LoopOver` / `Spawn` downstream, exactly like any other multi-result rule). With a uniquely-keyed lookup it is a single value.

A miss is visible, not silent. If no row matches the key (or a key value is NULL), the rule raises a recoverable per-row error naming the key — unless an optional final `default` value is given, in which case the default is returned on a miss. This mirrors `GetCDMTableId`.

PARAMETERS

1 — `table.return_field` *Mandatory* — accepts: table+field (`@table:field`)

The CDM table to read from and the field to return, e.g. `@procedure_occurrence:procedure_occurrence_id`.

2 — `keyFields` *Mandatory* — accepts: array of fields (`:-notation`)

Key fields on the looked-up table, e.g. `[:person_id, :procedure_source_value]`. Must not repeat a field.

3 — `values` *Mandatory* — accepts: array of values (fields or literals)

Values from the current row matched against `keyFields` position-by-position, e.g. `[:person_id, :proc_src]`.

4 — `default` *Mandatory* — accepts: any (optional)

Returned when the key matches no row (or a key value is NULL). Its type must match the return field. Omit it to make a miss a recoverable per-row error instead.

OUTPUT TYPE

The output type matches the type of the returned field; one result per matched row.

REQUIRES

- The looked-up table, the return field, and all key fields must exist.
- The key-field array and the value array must have the same (non-zero) length, with no repeated key field.
- A `default`, if given, must match the return field's type.

EXAMPLES

Single-key lookup

```
GetCDMValue(@person:person_id, [:person_source_value], [:patient_id])
```

Composite-key lookup

```
GetCDMValue(@procedure_occurrence:procedure_occurrence_id, [:person_id, :procedure_source_value], [:person_id, :proc_src])
```

With a default on a miss

```
GetCDMValue(@person:person_id, [:person_source_value], [:patient_id], 0)
```

SEE ALSO

GetCDMVisitId, GetCDMTableId, GetCDMFieldValue

GetCDMVisitId

aggregating | category: cdm-helpers | no `||`

Looks up a visit id on a named CDM table by key with the narrowest matching date interval.

Collects a visit id from a named table — typically `visit_occurrence` or `visit_detail`. The returned id is the one whose `start_date` / `end_date` interval most narrowly contains the start date (and optional end date) given as parameters.

PARAMETERS

Default key/value, single date

1 — `table` *Mandatory* — accepts: table

The CDM table the id is collected from, in `@`-notation. With this form the key field defaults to `<table_name>_source_value` and the value field to `<table_name>_id`.

2 — `keyValue` *Mandatory* — accepts: field

Field whose value must match the key field on the CDM table.

3 — `date` *Mandatory* — accepts: field

A date field whose value must be contained in the visit interval.

Default key/value, start/end date

1 — `table` *Mandatory* — accepts: table

The CDM table, in `@`-notation.

2 — `keyValue` *Mandatory* — accepts: field

Field whose value must match the key field on the CDM table.

3 — `startDate` *Mandatory* — accepts: field

Start date — the visit interval must contain this value.

4 — `endDate` *Mandatory* — accepts: field

End date — the visit interval must contain this value.

Explicit key/value fields

1 — `table` *Mandatory* — accepts: table

The CDM table, in `@`-notation.

2 — `key` *Mandatory* — accepts: field

The key field on the CDM table, in `:`-notation. Defaults to `<table_name>_source_value` if omitted.

3 — `value` *Mandatory* — accepts: field

The value field on the CDM table, in `:`-notation. Defaults to `<table_name>_id` if omitted.

4 — `keyValue` *Mandatory* — accepts: field

Field whose value must match the key field.

5 — `startDate` *Mandatory* — accepts: field

Start date of the interval to match.

6 — `endDate` *Mandatory* — accepts: field

End date of the interval to match.

INPUT TYPE

Parameter 1 is the CDM table; parameters 2 and 4 are field references of any matching type; parameters 5 and 6 are date fields.

OUTPUT TYPE

The output type matches the type of the value field.

EXAMPLES

Default fields, single date

```
GetCDMVisitId(@visit_occurrence, :recnum, :date)
```

Default fields, start and end date

```
GetCDMVisitId(@visit_occurrence, :recnum, :start_date, :end_date)
```

Explicit key/value with start and end date

```
GetCDMVisitId(@visit_occurrence, :visit_occurrence_source_value, :visit_occurrence_id, :recnum, :start_date, :end_date)
```

SEE ALSO

GetCDMTableId, GetCDMFieldValue

GetEnvironmentValue

aggregating | category: cdm-helpers | no `||` | resultSize: 1

Reads a value from the engine's environment database and returns it as a `STRING`.

`GetEnvironmentValue("KEY")` looks up a key in the engine's environment database and emits the associated value. The lookup happens once, at rule construction time, and the result is cached for the lifetime of the rule, so it has no per-row cost.

Use this for static configuration values that should be sourced from the engine's environment (e.g. default concept ids set per deployment) rather than baked into the DSL.

PARAMETERS

1 — `key` *Mandatory* — accepts: STRING

The environment-database key to look up.

INPUT TYPE

OUTPUT TYPE

- `STRING`

REQUIRES

- The key must exist in the engine's environment database.

EXAMPLES

Static gender concept id (test007)

```
GetEnvironmentValue("GENDER_CONCEPT")
```

Use a deployment-specific provider source

```
GetEnvironmentValue("EHR")
```

ERRORS

- `The environment key: 'KEY' wasn't found`

  The named key is not registered in the engine's environment database.

- `GetEnvironmentValue accepts exactly one string parameter`

  Wrong arity or non-string parameter.

- `` `GetEnvironmentValue' does not support multi-sized results ``

  The rule was used with a `[]` or `[N]` suffix.

SEE ALSO

SetValue, GetCDMTableId, GetCDMVisitId

GreaterThan

aggregating category: comparison no `||`

Returns TRUE when each parameter is strictly greater than the next.

Each value preceding another must be strictly greater than the next for the rule to produce a `TRUE` result. At least two parameters must be given.

PARAMETERS

N — `values` *Mandatory* — accepts: field, DATE, FLOAT, INT, STRING

The Nth value to compare against the next. May be a static value or a field reference.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.
- Two or more parameters must be given to the rule.

EXAMPLES

Three values

```
GreaterThan(:float1, :float2, 0.1)
```

Static followed by field

```
GreaterThan(12, :int)
```

SEE ALSO

GreaterThanOrEqual, LessThan, LessThanOrEqual, Equal, NotEqual

GreaterThanOrEqual

aggregating | category: comparison | no `||`

Returns TRUE when each parameter is greater than or equal to the next.

Each value preceding another must be greater than or equal to the next for the rule to produce a `TRUE` result. At least two parameters must be given.

PARAMETERS

N — `values` *Mandatory* — accepts: field, DATE, FLOAT, INT, STRING

The Nth value to compare against the next. May be a static value or a field reference.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.
- Two or more parameters must be given to the rule.

EXAMPLES

Three values

```
GreaterThanOrEqual(:float1, :float2, 0.1)
```

Static followed by field

```
GreaterThanOrEqual(12, :int)
```

SEE ALSO

GreaterThan, LessThan, LessThanOrEqual, Equal, NotEqual

Hours

aggregating | category: date | no `||`

Converts a number of hours into the corresponding number of seconds.

Helper that multiplies the input or parameter (in hours) into seconds.

PARAMETERS

1 — `hours` *Optional* — accepts: field, INT

Number of hours to convert to seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
Hours()
```

From static parameter

```
Hours(19)
```

SEE ALSO

Days, Minutes, Seconds, Months, Years, AsHours

IgnoreValue

aggregating | category: aggregation | no `||`

Removes a single input value from the rule logic context.

Drops the given value from the rule logic context so it is not visible to subsequent rules. Only one value can be removed per call and it must be provided as an input.

PARAMETERS

No parameters accepted.

INPUT TYPE

A single value of any type.

OUTPUT TYPE

No output is produced.

REQUIRES

- A single value as input.

EXAMPLES

Remove a named field from the context

```
:MyValueToIgnore => IgnoreValue()
```

SEE ALSO

Use, SetValue

LessThan

aggregating | category: comparison | no `||`

Returns TRUE when each parameter is strictly less than the next.

Each value preceding another must be strictly less than the next for the rule to produce a `TRUE` result. At least two parameters must be given.

PARAMETERS

N — `values` *Mandatory* — accepts: field, DATE, FLOAT, INT, STRING

The Nth value to compare against the next. May be a static value or a field reference.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.
- Two or more parameters must be given to the rule.

EXAMPLES

Three values

```
LessThan(:float1, :float2, 0.1)
```

Static followed by field

```
LessThan(12, :int)
```

SEE ALSO

LessThanOrEqual, GreaterThan, GreaterThanOrEqual, Equal, NotEqual

LessThanOrEqual

aggregating | category: comparison | no `||`

Returns TRUE when each parameter is less than or equal to the next.

Each value preceding another must be less than or equal to the next for the rule to produce a `TRUE` result. At least two parameters must be given.

PARAMETERS

N — `values` *Mandatory* — accepts: field, DATE, FLOAT, INT, STRING

The Nth value to compare against the next. May be a static value or a field reference.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.
- Two or more parameters must be given to the rule.

EXAMPLES

Three values

```
LessThanOrEqual(:float1, :float2, 0.1)
```

Static followed by field

```
LessThanOrEqual(12, :int)
```

SEE ALSO

LessThan, GreaterThan, GreaterThanOrEqual, Equal, NotEqual

Max

aggregating | category: aggregation | no `||`

Returns the largest value among the given parameters or inputs, folding ARRAY inputs element-by-element.

Without parameters, `Max` selects the maximum across all available inputs. When parameters are given, it chooses among those parameters only.

An ARRAY input is folded element-by-element — `Max` compares array *contents*, never an array as a whole — so scalars and arrays mix freely in one call (e.g. `Max(5, :int_array)`), and `GetCDMTableId(@idx, :key)[] => Max()` yields the maximum of all matched values. The element type drives the comparison; when every input is an ARRAY the output type defers to the destination field.

PARAMETERS

N — `candidates` *Optional* — accepts: field, DATE, FLOAT, INT, STRING

The Nth candidate value. May be a field reference (scalar or ARRAY) or a static value.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`
- `ARRAY`

OUTPUT TYPE

- `same-as-input`

REQUIRES

- All parameter types and (scalar) input types must be identical; ARRAY element types must match too.

EXAMPLES

Max across all inputs

```
Max()
```

Two field references

```
Max(:value1, :value2)
```

Field reference vs. static value

```
Max(:value, 42)
```

Lexicographic max of strings

```
Max(:text1, :text2, "Zulu")
```

Maximum of a collected ARRAY

```
GetCDMTableId(@idx, :key)[] => Max()
```

SEE ALSO

Min, Sum, Count, Avg, Selector

MergeDate

aggregating | category: date | no `||`

Combines several integer or date components into a single date.

Combines integers and dates labelled with their role into a single output date. Each parameter is a two-entry array: the first entry is the field reference, the second is the role.

The supported roles are:

- `date` (input type `DATE`)
- `time` (input type `DATE`)
- `year` (input type `INT`)
- `month` (input type `INT`)
- `day` (input type `INT`)
- `hour` (input type `INT`)
- `minute` (input type `INT`)
- `second` (input type `INT`)

PARAMETERS

1 — `component1` *Mandatory* — accepts: array

Two-entry array `[:field, "role"]` — the field to use and its role in the merged date.

2 — `component2` *Mandatory* — accepts: array

Two-entry array `[:field, "role"]` — the field to use and its role in the merged date.

N — `extras` *Optional* — accepts: array

Additional `[:field, "role"]` components, up to a total of six arrays.

INPUT TYPE

- `DATE`
- `INT`

OUTPUT TYPE

- `DATE`

REQUIRES

- Roles must be one of `date`, `time`, `year`, `month`, `day`, `hour`, `minute`, or `second`.

EXAMPLES

Combine a date with a separate time-of-day

```
MergeDate([:start_date, "date"], [:start_hour, "time"])
```

Combine year/month/day/hour/minute/second

```
MergeDate([:year, "year"], [:month, "month"], [:day, "day"], [:hour, "hour"], [:minute, "minute"], [:second, "second"])
```

SEE ALSO

DayFromDate, MonthFromDate, YearFromDate, HourFromDate, MinuteFromDate, SecondFromDate

Min

aggregating | category: aggregation | no `||`

Returns the smallest value among the given parameters or inputs, folding ARRAY inputs element-by-element.

Without parameters, `Min` selects the minimum across all available inputs. When parameters are given, it chooses among those parameters only.

An ARRAY input is folded element-by-element — `Min` compares array *contents*, never an array as a whole — so scalars and arrays mix freely in one call (e.g. `Min(5, :int_array)`), and `GetCDMTableId(@idx, :key)[] => Min()` yields the minimum of all matched values. The element type drives the comparison; when every input is an ARRAY the output type defers to the destination field.

PARAMETERS

N — `candidates` *Optional* — accepts: field, DATE, FLOAT, INT, STRING

The Nth candidate value. May be a field reference (scalar or ARRAY) or a static value.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`
- `ARRAY`

OUTPUT TYPE

- `same-as-input`

REQUIRES

- All parameter types and (scalar) input types must be identical; ARRAY element types must match too.

EXAMPLES

Min across all inputs

```
Min()
```

Two field references

```
Min(:value1, :value2)
```

Field reference vs. static value

```
Min(:value, 42)
```

Lexicographic min of strings

```
Min(:text1, :text2, "Alpha")
```

Minimum of a collected ARRAY

```
GetCDMTableId(@idx, :key)[] => Min()
```

SEE ALSO

Max, Sum, Count, Avg, Selector

Minutes

aggregating | category: date | no `||`

Converts a number of minutes into the corresponding number of seconds.

Helper that multiplies the input or parameter (in minutes) into seconds.

PARAMETERS

1 — `minutes` *Optional* — accepts: field, INT

Number of minutes to convert to seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
Minutes()
```

From static parameter

```
Minutes(240)
```

SEE ALSO

Days, Hours, Seconds, Months, Years, AsMinutes

Months

aggregating | category: date | no `||`

Converts a number of months into the corresponding number of seconds (30-day months).

Helper that multiplies the input or parameter (in months) into seconds. A single month is treated as 30 days, so 12 months is 360 days.

PARAMETERS

1 — `months` *Optional* — accepts: field, INT

Number of months to convert to seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.
- A single month is expected to be 30 days. Thus, 12 months is 360 days.

EXAMPLES

From input

```
Months()
```

From static parameter (12 months)

```
Months(12)
```

SEE ALSO

Days, Hours, Minutes, Seconds, Years, AsMonths

Multiply

aggregating category: arithmetic no `||`

Multiplies all values given to the rule, or all available inputs.

Multiplies all values supplied as parameters in order. If no parameters are given, all available inputs are multiplied. The output type is promoted to `FLOAT` if any input is a float; otherwise it is `INT`.

PARAMETERS

1 — `multiplier` *Optional* — accepts: field, INT, FLOAT

The first value of the multiplication. May be a field reference or a static value.

N — `multiplicands` *Optional* — accepts: field, INT, FLOAT

The Nth multiplicand. May be a field reference or a static value.

INPUT TYPE

- `FLOAT`
- `INT`

OUTPUT TYPE

- `matches-input-promotion`

Returns `FLOAT` if any input is a float, otherwise `INT`.

EXAMPLES

Multiply all available inputs

```
Multiply()
```

Multiplier and two multiplicands

```
Multiply(:multiplier, :multiplicand1, :multiplicand2)
```

Static values

```
Multiply(10, 10, 10)
```

Mixed static and field

```
Multiply(4.2, :multiplicand)
```

SEE ALSO

Add, Subtract, Divide

NotEqual

aggregating | category: comparison | no `||`

Returns TRUE when all values (parameters or inputs) are different.

Without parameters the rule checks that every input differs. With parameters the listed values — static or field references — must all be unequal to produce a `TRUE` result.

PARAMETERS

N — `values` *Optional* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

The Nth value to compare. May be a static value or a field reference. If no parameters are given, all available inputs are compared.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- All values must have the same type.

EXAMPLES

Compare all available inputs

```
NotEqual()
```

Three field references

```
NotEqual(:float1, :float2, :float3)
```

Mix static and field

```
NotEqual(17, :int)
```

SEE ALSO

Equal, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual

NumberOfResults

aggregating | category: cdm-helpers | no `||` | resultSize: 1

Returns the number of results produced by a sibling field, or by the current execution context.

`NumberOfResults` reports a row-level fan-out count as an `INT`. It has two forms:

- **No parameter** — `NumberOfResults()` returns `contextCache.currentResults`, the number of forks active in the current execution path. Useful for diagnostic side-channels and as a top-level row counter.
- **Field parameter** — `NumberOfResults(:other_field)` counts how many results the named sibling field produced. The other field must be in the same CDM table. In new mode, when the sibling holds an `ARRAY`, `NumberOfResults` returns the array's length directly.

For new-mode `ARRAY`-valued fields you can use the equivalent `Size()` rule, which reads the array length from a pipeline value rather than through a sibling-field dependency.

PARAMETERS

1 — `field` *Optional* — accepts: field

A sibling field reference. When omitted, the rule returns the number of results in the current execution context.

INPUT TYPE

OUTPUT TYPE

- `INT`

REQUIRES

- When given, the field reference must point to a field in the same CDM table.

EXAMPLES

Per-row count (test056)

```
NumberOfResults() => :result_count
```

Sibling-field result count

```
NumberOfResults(:year_of_birth) => :yb
```

New-mode ARRAY-aware count

```
NumberOfResults(:month_of_birth) => :mb
```

ERRORS

- `NumberOfResults requires a reference to a field in the same CDM table as the one the rule is executed from.`

  The parameter is not a valid sibling-field reference.

- `NumberOfResults accepts either no parameters or a single field reference parameter`

  Wrong arity.

- `` `NumberOfResults' does not support multi-sized results ``

  The rule was used with a `[]` or `[N]` suffix.

SEE ALSO

Size, GetCDMFieldValue, AutoGenId

Or

aggregating | category: logical | no `||`

Boolean OR across all parameters or all available inputs.

ORs together two or more boolean values. The values can be supplied as parameters; if no parameters are provided, all available inputs are ORed.

PARAMETERS

N — `operands` *Optional* — accepts: field, BOOLEAN

The Nth boolean to OR. May be a field reference (`:f`) or a static boolean. If no parameters are given, every available input is ORed.

INPUT TYPE

- `BOOLEAN`

OUTPUT TYPE

- `BOOLEAN`

REQUIRES

- At least two inputs are available.

EXAMPLES

OR all available inputs

```
Or()
```

OR of static booleans

```
Or(TRUE, FALSE)
```

OR of field references

```
Or(:BooleanValue1, :BooleanValue2)
```

SEE ALSO

And, Not

Seconds

aggregating | category: date | no `||`

Returns the input number of seconds unchanged — completes the time-unit set.

Helper companion to `Minutes`, `Hours`, `Days`, `Months`, and `Years`. Since the integer value is already in seconds the command simply passes the value through; it exists for symmetry with the other time-unit helpers.

PARAMETERS

1 — `seconds` *Optional* — accepts: field, INT

Number of seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.

EXAMPLES

From input

```
Seconds()
```

From static parameter

```
Seconds(53)
```

SEE ALSO

Days, Hours, Minutes, Months, Years, AsSeconds

Selector

aggregating | category: aggregation | supports `||`

Picks one of two values based on a boolean or a comparison between two values.

`Selector` operates in two modes — boolean and comparison.

In **boolean mode**, the first parameter is a boolean field reference. The second parameter is selected when the boolean is `TRUE`; the third (optional) parameter is selected when it is `FALSE`. Without a third parameter, `NULL` is used for the false branch.

In **comparison mode**, the first and third parameters are the values to compare. The second parameter is the comparison operator: `<`, `=`, `>`, `<=`, `<>`, or `>=`. The fourth parameter is selected when the comparison evaluates to `TRUE`; the fifth (optional) parameter is used for the false branch (defaults to `NULL`).

PARAMETERS

Boolean mode

1 — `condition` *Mandatory* — accepts: field

A boolean field reference whose value drives the selection.

2 — `trueValue` *Mandatory* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

Value chosen when the boolean is `TRUE`.

3 — `falseValue` *Optional* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

Value chosen when the boolean is `FALSE`. Defaults to `NULL` when omitted.

Comparison mode

1 — `left` *Mandatory* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

First value of the comparison.

2 — `operator` *Mandatory* — accepts: STRING

Comparison operator. One of `<`, `=`, `>`, `<=`, `<>`, `>=`.

3 — `right` *Mandatory* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

Second value of the comparison.

4 — `trueValue` *Mandatory* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

Value chosen when the comparison is `TRUE`.

5 — `falseValue` *Optional* — accepts: field, BOOLEAN, DATE, FLOAT, INT, STRING

Value chosen when the comparison is `FALSE`. Defaults to `NULL` when omitted.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

The output type matches the second parameter (boolean mode) or the fourth parameter (comparison mode).

REQUIRES

- In comparison mode the first and the third parameter must have the same type.
- If a third parameter (boolean mode) or fifth parameter (comparison mode) is present it must have the same type as the true-branch value.
- In comparison mode the operator must be one of `<`, `=`, `>`, `<=`, `<>`, `>=`.

EXAMPLES

Comparison mode — equal

```
Selector(:int1, "=", :int2, "Equal", "Unequal")
```

Comparison mode — less-than with default false branch

```
Selector(:str1, "<", "E", :str2)
```

Comparison mode — not-equal dates

```
Selector(:date1, "<>", :date2, "Unequal dates", "Equal dates")
```

Comparison mode — greater-than producing string

```
Selector(:float1, ">", :float2, :str1)
```

Boolean mode

```
Selector(:Boolean, "True", "False")
```

SEE ALSO

Equal, NotEqual, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual, Min, Max

SetValue

aggregating | category: aggregation | no `||`

Emits the supplied parameter value as the output.

Produces an output value drawn from the first parameter.

PARAMETERS

1 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, array

The value to emit.

INPUT TYPE

Does not accept input.

OUTPUT TYPE

The output type matches the type of the value parameter.

EXAMPLES

Static integer

```
SetValue(42)
```

Static string

```
SetValue("My Value")
```

SEE ALSO

Concat, IgnoreValue, Use

Size

aggregating | category: array | no `||` | resultSize: 1

Returns the number of elements in an `ARRAY` value as an `INT`.

`Size` consumes a single `ARRAY` pipeline input and returns its length as an `INT`. It is the array-mode analog of `NumberOfResults()` — whereas `NumberOfResults` counts row-level fan-out from multi-sized rules, `Size` reads the in-memory length of an `ARRAY` value directly.

Typically used in pipeline form: `:arr => Size() => :n` or chained after a producer such as `GetCDMFieldValue(:f)[]`.

PARAMETERS

No parameters accepted.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `INT`

REQUIRES

- Exactly one ARRAY-typed pipeline input.

EXAMPLES

Length of a collected ARRAY (test082)

```
GetCDMFieldValue(:year_of_birth)[] => Size() => :yb
```

Length of a previously-bound ARRAY field

```
:mob_arr => Size() => :mb
```

Project length straight to a string

```
GetCDMFieldValue(:year_of_birth)[] => Size() => AsString()
```

ERRORS

- `` `Size': input must be ARRAY-typed ``

  The pipeline value reaching the rule is not an `ARRAY`.

- `` `Size' requires exactly one ARRAY input ``

  Wrong number of pipeline inputs.

- `` `Size' does not accept parameters ``

  A parameter was supplied — the input must come from the pipeline.

SEE ALSO

NumberOfResults, At, Unique, Remove, Without

Spawn

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Expands an `ARRAY` value into individual rows, fanning the chain out to one row per element.

`Spawn(:array)` consumes an `ARRAY` field and emits one row per element of the array. The first element becomes the primary result; the remaining elements become additional results, so downstream rules in the chain run once per element.

This is the new-mode replacement for the legacy multi-sized result fan-out: `Spawn` lets you defer fan-out from the producer (`UsagiMap()[]`, `Use()[N]`, ...) to a later, explicit step that operates on an `ARRAY` collected up to that point. Available in new mode only.

PARAMETERS

1 — `array` *Mandatory* — accepts: field

Field reference to an `ARRAY`-typed input. Each element of the array becomes one row in the downstream chain.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

The element type of the input `ARRAY`. The rule reports `ARRAY` from `aggregateType()` purely to bypass the parse-time type check; elements are typed at runtime.

REQUIRES

- Exactly one parameter, a field reference to an `ARRAY` field.
- The `ARRAY` must contain at least one element at runtime — empty arrays raise a runtime error.

EXAMPLES

Expand a collected ARRAY into one row per element

```
:codes => UsagiMap("file.csv")[] => :concept_array
Spawn(:concept_array)
```

ERRORS

- `` `Spawn': input must be ARRAY-typed ``

  The referenced field does not hold an `ARRAY` value at runtime.

- `` `Spawn': empty array - no rows to produce ``

  The `ARRAY` had zero elements when the rule fired.

- `` Spawn: input :f' must be of ARRAY type` ``

  The referenced field is not declared `ARRAY`.

- `` `Spawn' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

Use, CartesianOf, Unique, At, Size

Subtract

aggregating | category: arithmetic | no `||`

Subtracts the remaining values from the first, with date- and float-aware promotion.

Subtracts each subsequent parameter from the first. With no additional parameters, the first value is reduced by all available inputs. Date arithmetic is supported: a single date in the input keeps the output as a date, while two dates produce an integer (their difference in seconds).

PARAMETERS

1 — `minuend` *Mandatory* — accepts: field, INT, FLOAT, DATE

The value to subtract from. May be a field reference or a static value.

N — `subtrahends` *Optional* — accepts: field, INT, FLOAT, DATE

The Nth value to subtract from the minuend. May be a field reference or a static value.

INPUT TYPE

- `INT`
- `FLOAT`
- `DATE`

OUTPUT TYPE

- `matches-input-promotion`

Returns `FLOAT` if any input is a float; `DATE` if a single date is in the input; `INT` if two dates are in the input.

REQUIRES

- At most two dates can be given in the input.
- A float cannot be subtracted from a date.
- A date can neither be subtracted from a float nor an integer.
- If a float is in the input the return type will always be a float.
- If a single date is in the input, the return type will be a date.
- If two dates are in the input, the return type will be an integer.

EXAMPLES

Subtract from all available inputs

```
Subtract(42)
```

Subtract a static value from a field

```
Subtract(:float, 42)
```

Date difference in seconds

```
Subtract(:date1, :date2)
```

Multiple subtrahends

```
Subtract(:integer, 1000, :float)
```

SEE ALSO

Add, Multiply, Divide, Days, Hours, Minutes, Seconds

Sum

aggregating category: aggregation no `||`

Returns the sum of the given parameters or the available inputs, folding ARRAY inputs element-by-element.

Without parameters, `Sum` adds every available input. When parameters are given, it adds only those parameters/fields. Scalars and ARRAY values mix freely in one call — an ARRAY input is folded element-by-element (the same way `Min` / `Max` fold arrays), so `GetCDMTableId(@idx, :key)[] => Sum()` totals all matched values.

NULL inputs and NULL/empty ARRAY elements are skipped, never poisoning the total. Output is `INT` when every value is an integer, otherwise `FLOAT`. When every input is an ARRAY the element type is unknown at parse time, so the output type defers to the destination field.

`Sum` accepts only numeric (`INT` / `FLOAT`) values; dates and strings are rejected at setup.

PARAMETERS

N — `addends` *Optional* — accepts: field, FLOAT, INT

The Nth value to add. May be a field reference (scalar or ARRAY) or a static number.

INPUT TYPE

- `FLOAT`
- `INT`
- `ARRAY`

OUTPUT TYPE

- `INT when all values are integers, otherwise FLOAT`

REQUIRES

- All parameter and input types must be numeric (INT or FLOAT).

EXAMPLES

Sum across all inputs

```
Sum()
```

Two field references

```
Sum(:value1, :value2)
```

Total a collected ARRAY

```
GetCDMTableId(@line_index, :claim_id, :amount)[] => Sum()
```

SEE ALSO

Add, Min, Max, Count

Tuple

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Packs two or more named fields into a single `TUPLE` value (or `ARRAY<TUPLE>` when any input is an `ARRAY`).

`Tuple(:f1, :f2, ...)` builds a `TUPLE` value from the supplied field references, in declared order. The resulting tuple is typically stored in a virtual field and later disassembled with `TupleGet` — this lets multiple values flow through a single pipeline slot, useful for `||`-fallback blocks that must produce more than one named output.

When any input field holds an `ARRAY`, `Tuple` broadcasts: it emits one `TUPLE` per `ARRAY` element, returning an `ARRAY<TUPLE>` whose length matches the input array's length. Scalar inputs are repeated across every tuple. Empty input `ARRAY`s raise a `RuntimeMapError` so that an enclosing `||` fallback can try the next block.

Available in new mode only.

PARAMETERS

N — `field` *Mandatory* — accepts: field

Two or more field references in `:`-notation. Each referenced field must exist in the input set at parse time.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`
- `ARRAY`

OUTPUT TYPE

- `TUPLE`

An `ARRAY<TUPLE>` when any input field holds an `ARRAY`.

REQUIRES

- At least two field-reference parameters.
- Every referenced field must be present in the rule's input set at parse time.

EXAMPLES

Pair two related concept ids for `||`-fallback (test092)

```
{ :gender => UsagiMap("map3.csv")[] => :gid
  :ethnicity => UsagiMap("eth.csv")[] => :eth
  Tuple(:gid, :eth) => :gtuple
}
|| { ... }
TupleGet(:gtuple, 0) => :gender_concept_id
TupleGet(:gtuple, 1) => :ethnicity_concept_id
```

ERRORS

- `Tuple: input has no value (field mapping failed with no default)`

  A referenced field has no value at runtime — the enclosing `||` fallback should pick this up.

- `Tuple: empty ARRAY input, no tuples produced`

  A broadcasted `ARRAY` input was empty.

- `Tuple() requires at least two field parameters in :-notation`

  Fewer than two parameters were supplied.

- `Tuple(): field 'X' not found in input`

  A named field reference does not match any input field.

- `Tuple() is only available in new mode`

  New mode is not enabled in `etl.conf`.

SEE ALSO

TupleGet, Use, Selector

TupleGet

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Extracts the element at the given 0-based index from a `TUPLE`-valued field.

`TupleGet(:tuple, N)` reads a previously-built `TUPLE` value from the named field and returns its `N` th element. It does not consume any pipeline input; the dependency is declared via the field reference, so `TupleGet` can appear at the start of a sibling chain that needs to unpack values produced earlier by a `Tuple` rule.

When the source `TUPLE` was broadcast from an `ARRAY` (i.e. the field holds an `ARRAY<TUPLE>`), `TupleGet` propagates the fan-out: the `N` th element is extracted from each `TUPLE` in the array, with one result per element.

Available in new mode only.

PARAMETERS

1 — `tuple` *Mandatory* — accepts: field

Field reference (`:`-notation) to a CDM field holding a `TUPLE` value (or an `ARRAY<TUPLE>`).

2 — `index` *Mandatory* — accepts: INT

0-based index into the tuple. Must be non-negative.

INPUT TYPE

OUTPUT TYPE

- `UNKNOWN`

The element type is inferred from the destination CDM field. Out-of-range indices raise a runtime error.

REQUIRES

- First parameter must be a field reference in :-notation.
- Second parameter must be a non-negative integer index.
- The referenced field must hold a `TUPLE` (or `ARRAY<TUPLE>`) at runtime.

EXAMPLES

Unpack a tuple stored in a virtual field (test092)

```
Tuple(:cid, :eth) => :gtuple
TupleGet(:gtuple, 0) => :gender_concept_id
TupleGet(:gtuple, 1) => :ethnicity_concept_id
```

ERRORS

- `TupleGet: dependency value is not a TUPLE`

  The referenced field does not hold a `TUPLE` at runtime.

- `TupleGet: index out of range`

  The 0-based index is `>=` the tuple length.

- `TupleGet() index must be non-negative`

  A negative index literal was supplied.

- `TupleGet() requires exactly two parameters`

  Wrong arity.

- `TupleGet() is only available in new mode`

  New mode is not enabled in `etl.conf`.

SEE ALSO

Tuple, GetCDMFieldValue

Unique

aggregating | category: aggregation | no `||` | resultSize: N

Deduplicates values from one or more inputs; in new mode also flattens `ARRAY` inputs into one `ARRAY`.

`Unique` operates in two modes depending on the input value type.

Scalar mode (any execution mode). With non-`ARRAY` inputs `Unique` runs as a multi-sized aggregator: the first unique value seen across all input fields becomes the primary result, every subsequent unique value is emitted as an additional result, and duplicates are dropped. A per-rule `seen` set is shared across the row's forks via `contextCache.uniqueValuesByRule`, making `Unique` a useful convergence point after a multi-sized fan-out.

Array mode (new mode). When any input field is `ARRAY`-typed, all inputs must be `ARRAY` and `Unique` flattens them, deduplicates the combined elements, and returns a single `ARRAY` value rather than fanning out into rows.

Without parameters `Unique` consumes every pipeline input. With `Unique(:f1, :f2, ...)` it considers only the named fields. All scalar-mode inputs must share the same value type.

PARAMETERS

N — `field` *Optional* — accepts: field

Field references in `:`-notation. When omitted, every pipeline input is considered.

INPUT TYPE

- `INT`
- `FLOAT`
- `DATE`
- `STRING`
- `ARRAY`

OUTPUT TYPE

Scalar mode: type matches the (shared) input type. Array mode: `ARRAY` containing the unique elements.

REQUIRES

- Scalar mode: all inputs must share the same value type (one of INT, FLOAT, DATE, STRING).
- Array mode: all inputs must be ARRAY-typed; only available in new mode.

EXAMPLES

Multi-sized convergence over two pipeline values (test062)

```
Unique(:var1, :var2)[]
```

Use across all pipeline inputs

```
Unique()[]
```

Array mode — flatten and dedup, then map per element (test085)

```
Unique(:resultA, :resultD)[] | AsString()
```

ERRORS

- `Unique requires all input fields to have the same type`

  Scalar-mode inputs of mixed types — combine like-typed inputs only.

- `Unique accepts date, float, integer, and string input only`

  A scalar input of an unsupported type (e.g. `BOOLEAN`).

- `Unique (array mode): all input fields must be ARRAY-typed`

  A mix of `ARRAY` and scalar inputs in array mode.

- `No unique values found - all input values are duplicates`

  Every value seen by the rule had already been emitted on this row.

- `Unique with ARRAY inputs is only available in new mode`

  An `ARRAY` field was passed but new mode is disabled.

SEE ALSO

Without, Append, Size, Use, CartesianOf

Use

aggregating | category: aggregation | no `||` | resultSize: N

Selects which field or local variable becomes the output of the rule chain.

When multiple source fields map to a CDM field, or several local field variables exist, `Use` selects which one is committed. `Use` always removes the input fields and leaves only one output, even when an output field is explicitly defined. `Use` allows multi-sized results.

Writing `Use(:Result) => :Result` keeps the named field in the rule chain so it remains addressable as `:Result` downstream.

PARAMETERS

1 — `primary` *Mandatory* — accepts: field

The field or local field variable to commit to the CDM field.

N — `extras` *Optional* — accepts: field

Additional fields or local field variables — used when committing to a multi-sized CDM field.

INPUT TYPE

- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

Same type as the input — array-shaped when used with `[]`.

REQUIRES

- Allows multi-sized results.

EXAMPLES

Single result

```
Use(:Result)
```

Multi-sized result

```
Use(:Result1, :Result2, :Result3)[]
```

Use a CDM field

```
Use(:location_id)
```

Keep the result accessible downstream

```
Use(:Result) => :Result
```

Leaves `:Result` in the rule chain so it can still be referenced.

SEE ALSO

SetValue, IgnoreValue, Selector

Without

aggregating | category: array | no `||` | new-mode: required | resultSize: 1

Set difference of two `ARRAY` inputs — elements of `:source` not present in `:exclude`.

`Without` is an array-aware aggregating rule that returns the elements of its first input that are not equal to any element of its second input. Element comparisons follow the per-type `Value` equality operator. The result is a single `ARRAY` whose length is at most the source length.

Two equivalent shapes are accepted:

- **Parameter form** — `Without(:source, :exclude)` resolves the two inputs by field name.
- **Pipeline form** — `:source, :exclude => Without()` uses the two positional pipeline inputs.

Available in new mode only.

PARAMETERS

Parameter form

1 — `source` *Mandatory* — accepts: field

Field reference to the source `ARRAY`.

2 — `exclude` *Mandatory* — accepts: field

Field reference to the `ARRAY` whose elements are excluded.

Pipeline form (no parameters)

No parameters accepted.

INPUT TYPE

- `ARRAY`

OUTPUT TYPE

- `ARRAY`

REQUIRES

- Both inputs must be ARRAY-typed.
- Pipeline form requires exactly two pipeline inputs.

EXAMPLES

Pipeline form

```
:source, :exclude => Without() => :result
```

Parameter form

```
Without(:all_codes, :blocked) => :allowed
```

ERRORS

- `` `Without': source input must be ARRAY-typed ``

The first input value is not an `ARRAY` at runtime.

- `` `Without': exclude input must be ARRAY-typed ``

  The second input value is not an `ARRAY` at runtime.

- `` `Without' parameters must be field references ``

  A non-field-reference parameter was supplied.

- `` `Without' with no parameters requires exactly two ARRAY pipeline inputs ``

  The pipeline form was used with the wrong number of inputs.

- `` `Without' is only available in new mode ``

  New mode is not enabled in `etl.conf`.

SEE ALSO

Append, Unique, Remove, Size, CartesianOf

Years

aggregating | category: date | no `||`

Converts a number of years into the corresponding number of seconds (365-day years).

Helper that multiplies the input or parameter (in years) into seconds. A single year is treated as 365 days.

PARAMETERS

1 — `years` *Optional* — accepts: field, INT

Number of years to convert to seconds. If omitted, a single integer input must be provided.

INPUT TYPE

- `INT`

OUTPUT TYPE

- `INT`

REQUIRES

- Either a single value must be presented as an input or a single integer parameter must be provided.
- A single year is expected to be 365 days.

EXAMPLES

From input

```
Years()
```

From static parameter

```
Years(2)
```

SEE ALSO

Days, Hours, Minutes, Seconds, Months, AsYears

5.1.4 Table Rules

Table rules

Table rules act on entire CDM tables (joins, normalisation, dependencies).

TABLE-CONTROL

- DependOn — Declares that the current CDM table must not be transformed before another named table is loaded.
- InsertRecord — Inserts a static record into the current CDM table.
- Join — Builds merged records from two source tables, optionally restricted to matching keys.
- LeftJoin — Like `Join`, but always emits every record from the left table even when no key match is found on the right.
- LoopOver — Declares how multi-sized result fields produce CDM rows — zipping or cartesian product.
- Normalize — Builds a unique-key index on the CDM table; duplicate records are silently skipped.
- StoreIndexMap — Persists an index from the current CDM table to disk so a later run can reuse it.
- StoreSequence — Persists a global ID sequence to disk so a later run can continue it.
- TraceWhen — Enables per-record verbose logging for source rows whose named field matches one of a list of values.

DependOn

table | category: table-control | no `||` | resultSize: 1

Declares that the current CDM table must not be transformed before another named table is loaded.

Use `DependOn` to express a load-order requirement that the engine cannot infer from the Rabbit-in-a-Hat model. The most common case is a foreign-key relationship in the destination database: the engine does not see those constraints, so a `DependOn` rule on the dependent table is what guarantees the parent table is fully loaded first.

Beware that `DependOn` can introduce cycles between CDM tables. The engine does not break cycles automatically; if a cycle exists, neither table is loaded.

PARAMETERS

1 — `table` *Mandatory* — accepts: table reference (`@-notation`)

The CDM table that the current table depends on.

REQUIRES

- The referenced CDM table must exist.

EXAMPLES

```
DependOn(@location)
```

SEE ALSO

Join, LeftJoin

InsertRecord

table | category: table-control | no `||` | resultSize: 1

Inserts a static record into the current CDM table.

Useful when a CDM table needs a fixed set of rows that are not derived from the source data. The most common case is the `cdm_source` table, which typically contains a single row describing the data source.

Each parameter is a keyword-style assignment whose key matches a field name on the current CDM table and whose value matches that field's type. All non-nullable fields must be supplied.

PARAMETERS

N — `assignments` *Mandatory* — accepts: keyword parameters (`field=value`)

One `field=value` pair per field to populate. The field name must exist on the table; the value's type must match the field's type. Every non-nullable field must be provided.

REQUIRES

- All non-nullable fields on the table must be assigned a value.
- Each assignment's value type must match its field's type.

EXAMPLES

A minimal person row

```
InsertRecord(person_id=1, gender_concept_id=8507, year_of_birth=2000, race_concept_id=4218674, ethnicity_concept_id=38003564, location_id=1)
```

cdm_source metadata

```
InsertRecord(cdm_source_name="Synthetic Massachusetts",
  cdm_source_abbreviation="Synthmas",
  cdm_holder="Rook IT ApS",
  source_description="Test translation of the Synthmas dataset",
  source_release_date='2023-01-13',
  cdm_release_date='2023-06-30',
  cdm_version="5.4",
  vocabulary_version="v5.0 31-MAY-23")
```

Join

`table` category: table-control no `||` resultSize: 1

Builds merged records from two source tables, optionally restricted to matching keys.

`Join` produces all permutations of records from the two input tables unless keys are supplied — when keys are given, the join is restricted to records whose key values match. The semantics mirror an SQL inner join, though the syntax differs.

By default each key pair is matched by equality. An optional comparison array (one operator per key pair) makes a join a **range join**: each pair is tested with its own operator and the whole set is AND-ed together, like an SQL `ON` clause. Operators are `"="`, `">"`, `">="`, `"<"`, `"<="`. Omitting the array means all `"="` — exactly the equality behaviour above. Equality pairs drive the index lookup; comparison pairs are applied as filters on the matched candidates. A join with *no* `"="` pair at all is a pure-range join (the right table is scanned and the comparisons do all the matching). The comparison sides must be field references — a constant bound is a filter, not a join condition, so it is not accepted here.

Multiple `Join` / `LeftJoin` rules can be chained by naming the result table (the optional trailing name parameter); subsequent joins reference that name as a regular `@table`.

PARAMETERS

Cartesian (no keys)

1 — `left` *Mandatory* — accepts: table reference

2 — `right` *Mandatory* — accepts: table reference

Keyed

1 — `left` *Mandatory* — accepts: table reference

2 — `right` *Mandatory* — accepts: table reference

3 — `leftKey` *Mandatory* — accepts: array of fields

Key fields on the left table; must have the same length as `rightKey`.

4 — `rightKey` *Mandatory* — accepts: array of fields

Key fields on the right table; must have the same length as `leftKey`.

Keyed and named

1 — `left` *Mandatory* — accepts: table reference

2 — `right` *Mandatory* — accepts: table reference

3 — `leftKey` *Mandatory* — accepts: array of fields

4 — `rightKey` *Mandatory* — accepts: array of fields

5 — `name` *Mandatory* — accepts: STRING

A name for the joined result table; can be referenced from later `Join` / `LeftJoin` rules.

Range (comparison operators)

1 — `left` *Mandatory* — accepts: table reference

2 — `right` *Mandatory* — accepts: table reference

3 — `leftKey` *Mandatory* — accepts: array of fields

4 — `rightKey` *Mandatory* — accepts: array of fields

5 — `operators` *Mandatory* — accepts: array of STRING

One comparator per key pair — `"="`, `">"`, `">="`, `"<"`, `"<="`. AND-ed together. Omit for all-equality.

6 — `name` *Optional* — accepts: STRING

Optional result-table name (must come after the operator array).

REQUIRES

- All referenced tables and fields must exist.
- The same number of keys must exist for both tables.
- The operator array, if present, must have one entry per key pair.
- Key fields on the two sides must have matching types.
- Comparison operands must be fields (constants are filters, not joins).
- The named result table must not already exist.

EXAMPLES

```
Join(@conditions.csv, @encounter.csv, [:encounter], [:id], "ce_temp")
```

```
Join(@encounter.csv, @conditions.csv)
```

```
Join(@observations.csv, @patients.csv, [:patient], [:id])
```

Range join — equality on one key, a comparison on another

```
Join(@a.csv, @b.csv, [:k, :date], [:bk, :date], ["=", ">"])
```

Pure-range join — no equality key, value between two bounds

```
Join(@a.csv, @b.csv, [:lo, :hi], [:val, :val], ["<=", ">="])
```

SEE ALSO

LeftJoin

LeftJoin

table category: table-control no `||` resultSize: 1

Like `Join`, but always emits every record from the left table even when no key match is found on the right.

`LeftJoin` requires keys for both tables (unlike `Join`, which can run un-keyed). The left table's records are always preserved; right-table fields are filled in only where a key match exists.

An optional comparison array (one operator per key pair — `"="`, `">"`, `">="`, `"<"`, `"<="`, AND-ed together) turns this into a range left join, exactly as for `Join`. Every left record is still preserved; the comparisons decide which right records (if any) match. Omit the array for all-equality.

PARAMETERS

1 — `left` *Mandatory* — accepts: table reference

2 — `right` *Mandatory* — accepts: table reference

3 — `leftKey` *Mandatory* — accepts: array of fields

Key fields on the left table. Must have at least one entry and the same length as `rightKey`.

4 — `rightKey` *Mandatory* — accepts: array of fields

Key fields on the right table.

5 — `operators` *Optional* — accepts: array of STRING

Optional comparator per key pair (`"="`, `">"`, `">="`, `"<"`, `"<="`). Omit for all-equality.

6 — `name` *Optional* — accepts: STRING

A name for the joined result table (after the operator array, if present); can be referenced from later `Join` / `LeftJoin` rules.

REQUIRES

- All referenced tables and fields must exist.
- The same number of keys must exist for both tables.
- The operator array, if present, must have one entry per key pair.
- Key fields on the two sides must have matching types.
- At least one key must be supplied for each table.
- Comparison operands must be fields (constants are filters, not joins).
- The named result table must not already exist.

EXAMPLES

```
LeftJoin(@patients.csv, @conditions.csv, [:id, :deathdate], [:patient, :start], "pc_temp")
```

```
LeftJoin(@patients.csv, @conditions.csv, [:id], [:patient])
```

Range left join — keep every left row, comparison decides matches

```
LeftJoin(@a.csv, @b.csv, [:k, :date], [:bk, :date], ["=", ">"])
```

SEE ALSO

Join

LoopOver

table category: table-control no `||` resultSize: 1

Declares how multi-sized result fields produce CDM rows — zipping or cartesian product.

Any CDM-table mapping that uses a multi-sized result (`[]` or `[N]` on a rule) must declare a `LoopOver` so the engine knows how to combine the multiple values:

- Fields placed in the **same array** are zipped — the i-th result of one is paired with the i-th result of the others.
- Fields placed in **different arrays** form a cartesian product.

Within a single zipped group, every field must produce the same number of results at runtime. If the `[]` rule is unbounded, every field in the group must also be unbounded; if the rule is bounded with `[N]`, every field in the group must use the same `N`.

Even a CDM table with only one multi-sized field must declare it via `LoopOver`.

PARAMETERS

N — `group` *Mandatory* — accepts: array of fields (`:-notation`)

The Nth zipped group. Within the array, all fields are paired by index. Multiple `group` arrays form a cartesian product.

REQUIRES

- Every referenced field must produce a multi-sized result.
- Bounded multi-sized fields within a group must share the same bound.
- Unbounded multi-sized fields cannot be grouped with bounded ones.

EXAMPLES

Single multi-sized field

```
LoopOver([:field_one])
```

Two zipped fields

```
LoopOver([:field_one, :field_two])
```

Two zipped fields × one cartesian field

```
LoopOver([:field_one, :field_two], [:cartesian_field_three])
```

SEE ALSO

Spawn, Unique

Normalize

table category: table-control no `||` resultSize: 1

Builds a unique-key index on the CDM table; duplicate records are silently skipped.

`Normalize` declares a uniqueness constraint over a set of CDM-table fields. As records are inserted, the engine checks them against the index and discards any that would duplicate an existing key. This is essential when several source tables map into the same CDM table and may produce overlapping records (a typical case is the `location` table).

Even when only one source table maps in, normalisation is often worthwhile — for example, `provider` records frequently share an address.

At most one `Normalize` rule per CDM table.

PARAMETERS

1 — `key` *Mandatory* — accepts: array of fields (`:-notation`)

The fields whose combined value must be unique on the table.

REQUIRES

- All referenced CDM fields must exist.
- At least one field must be supplied.

EXAMPLES

```
Normalize([:address_1, :city, :state, :zip])
```

StoreIndexMap

table category: table-control no `||` resultSize: 1

Persists an index from the current CDM table to disk so a later run can reuse it.

Storing an index map lets you split an ETL across multiple Rabbit-in-a-Hat files. After a run, the named index and its mapped values are serialised to the directory configured under `rabbit configuration.index directory`. A subsequent run loads it automatically, which is essential when the second run uses `GetCDMTableId` to map source IDs to CDM IDs assigned by the first run.

PARAMETERS

1 — `index` *Mandatory* — accepts: array of fields (`:-notation`)

Fields that constitute the index key.

2 — `values` *Mandatory* — accepts: array of fields (`:-notation`)

Fields whose values are stored against the index.

REQUIRES

- All referenced CDM fields must exist.

EXAMPLES

```
StoreIndexMap([:person_source_value], [:person_id])
```

SEE ALSO

StoreSequence

StoreSequence

table category: table-control no `||` resultSize: 1

Persists a global ID sequence to disk so a later run can continue it.

Used in tandem with `AutoGenId` when populating a CDM table over several runs. After a run, the named global sequence is serialised; the next run picks up where it left off, ensuring generated IDs remain unique across runs.

PARAMETERS

1 — `name` *Mandatory* — accepts: STRING

The name of an existing global sequence created with `AutoGenId`.

REQUIRES

- A global sequence with the given name must already exist.

EXAMPLES

```
StoreSequence("drug_exposure_seq")
```

SEE ALSO

StoreIndexMap, AutoGenId

TraceWhen

table category: table-control no `||` resultSize: 1

Enables per-record verbose logging for source rows whose named field matches one of a list of values.

`TraceWhen` is a targeted-debugging rule. When a source row's named field matches any value in the supplied list, every rule that runs for that row writes a detailed trace line to the per-table log — even when the global log level is INFO.

Tracing has a per-row cost only for matching rows; non-matching rows go through the normal fast path. It is safe to leave a `TraceWhen` rule in place during a long run as long as the trigger list is small.

Tracing is automatically suppressed inside recursive `additionalQueue` mapping calls to prevent re-entrant logging from corrupting the trace.

PARAMETERS

1 — `source_field` *Mandatory* — accepts: field reference (`:-notation`)

The source field whose value is matched against the trigger list.

2 — `triggers` *Mandatory* — accepts: array of values

Values that activate per-row tracing. Element types must be compatible with `source_field` 's type.

REQUIRES

- The named source field must exist on the source table.

EXAMPLES

Trace a small set of patient IDs

```
TraceWhen(:patient_id, ["abc-123", "def-456"])
```

5.1.5 Filter Rules

Filter rules

Filter rules pass through a subset of records by group.

FILTER

- First — Passes through only the records with the lowest ordering value within each group.
- Last — Passes through only the records with the highest ordering value within each group.

First

filter | category: filter | no `||` | resultSize: 1

Passes through only the records with the lowest ordering value within each group.

`First` is a source-table filter rule. It groups source records by one set of fields and, within each group, lets through the single record with the smallest value across a second set of ordering fields.

PARAMETERS

1 — `group` *Mandatory* — accepts: array of fields (`@table:field` notation)

Defines how the source records must be grouped. All records that share the same values for every field in this array belong to the same group.

2 — `order` *Mandatory* — accepts: array of fields (`@table:field` notation)

Defines how to select the first value within each group. The record with the lowest collective value across these fields is the one passed through.

REQUIRES

- All referenced fields must exist on the source table.

EXAMPLES

```
First([:person_id],[:move_date])
```

SEE ALSO

Last

Last

filter | category: filter | no `||` | resultSize: 1

Passes through only the records with the highest ordering value within each group.

`Last` is the mirror of `First`: same grouping/ordering shape, but the record with the *largest* collective value across the ordering fields is the one let through.

PARAMETERS

1 — `group` *Mandatory* — accepts: array of fields (`@table:field` notation)

Defines how the source records must be grouped.

2 — `order` *Mandatory* — accepts: array of fields (`@table:field` notation)

Defines how to select the last value within each group. The record with the highest collective value across these fields is the one passed through.

REQUIRES

- All referenced fields must exist on the source table.

EXAMPLES

```
Last([:person_id],[:move_date])
```

SEE ALSO

First

5.1.6 Control-Flow Rules

Control rules

Control-flow rules direct execution within a rule chain.

CONTROL-FLOW

- Break — Aborts the current execution path so the surrounding row is skipped.
- JumpIf — Jumps to a named `Mark` when the input value equals the given parameter.
- JumpIfNot — Jumps to a named `Mark` when the input value does NOT equal the given parameter.
- JumpTo — Unconditionally jumps to a named `Mark` later in the rule chain.
- Mark — Declares a labelled jump target reachable by `JumpIf`, `JumpIfNot`, and `JumpTo`.

Break

control category: control-flow no `||` resultSize: 1

Aborts the current execution path so the surrounding row is skipped.

`Break` discards the current execution context: no row is emitted for the path on which it fires. It is typically placed inside a conditional branch (`If/Else`) or inside a multi-sized fan-out where some forks should drop without producing output.

`Break` accepts no parameters and no inputs, and produces no value. The surrounding chain machinery treats the path as terminated.

PARAMETERS

No parameters accepted.

INPUT TYPE

OUTPUT TYPE

No output — terminates the current execution path.

REQUIRES

- No parameters are accepted.

EXAMPLES

Drop a forked path that fails an assertion

```
If (:concept_id == 0) {
  Break()
}
```

SEE ALSO

JumpTo, JumpIf, JumpIfNot, Mark, AssertNot

JumpIf

control category: control-flow no `||` resultSize: 1

Jumps to a named `Mark` when the input value equals the given parameter.

`JumpIf` is part of the imperative control-flow cluster. It accepts a single pipeline input and compares it against a literal value. When the comparison succeeds, execution skips ahead to the `Mark(...)` rule with the matching name; otherwise the chain continues normally. The input value is passed through unchanged regardless of whether the jump fires.

Use it to short-circuit branches that do not need further processing — e.g. when a default has already been chosen and downstream lookups are unnecessary.

PARAMETERS

1 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL

The literal compared against the input value. Its type must match the input's type (or be `NULL`).

2 — `mark` *Mandatory* — accepts: STRING

The name of the destination `Mark` to jump to.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `same-as-input`

The input value is passed through unchanged.

REQUIRES

- Exactly one pipeline input.
- Parameter type must match the input type (or be NULL).
- The destination `Mark` must exist in the same chain.

EXAMPLES

Jump past default branch when concept already resolved

```
:visit_concept => JumpIfNot(9202, "Default") => :visit_concept
```

Used together with `JumpTo` and `Mark` to form an if/else (see test053).

ERRORS

- `JumpIf got too many inputs`

  More than one value reached the rule. `JumpIf` accepts a single input.

- `JumpIf accepts input of types: INTEGER, STRING, FLOAT, DATE, or BOOLEAN`

  An unsupported input type (e.g. ARRAY) was supplied.

- `JumpIf requires the type of the input and the type of the parameter to be identical`

  The literal value's type does not match the input field's declared type.

SEE ALSO

JumpIfNot, JumpTo, Mark

JumpIfNot

control category: control-flow no `||` resultSize: 1

Jumps to a named `Mark` when the input value does NOT equal the given parameter.

Inverse of `JumpIf`. When the input differs from the supplied literal, execution skips to the named `Mark`; otherwise the chain continues normally. The input value is passed through unchanged in both cases.

Combined with `JumpTo` and `Mark`, `JumpIfNot` implements an if/else pattern in the rule chain (see the imperative branching cluster).

PARAMETERS

1 — `value` *Mandatory* — accepts: BOOLEAN, DATE, FLOAT, INT, STRING, NULL

The literal compared against the input value. Its type must match the input's type (or be `NULL`).

2 — `mark` *Mandatory* — accepts: STRING

The name of the destination `Mark` to jump to.

INPUT TYPE

- `BOOLEAN`
- `DATE`
- `FLOAT`
- `INT`
- `STRING`

OUTPUT TYPE

- `same-as-input`

The input value is passed through unchanged.

REQUIRES

- Exactly one pipeline input.
- Parameter type must match the input type (or be NULL).
- The destination `Mark` must exist in the same chain.

EXAMPLES

If/else chain — skip the default branch when concept matches

```
:visit_concept => JumpIfNot(9202, "Default") => :visit_concept
// ...processing for the matching case...
JumpTo("Done")
Mark("Default")
// ...default-branch processing...
Mark("Done")
```

ERRORS

- `JumpIfNot got too many inputs`

  More than one value reached the rule. `JumpIfNot` accepts a single input.

- `JumpIfNot accepts input of types: INTEGER, STRING, FLOAT, DATE, or BOOLEAN`

  An unsupported input type (e.g. ARRAY) was supplied.

- `JumpIfNot requires the type of the input and the type of the parameter to be identical`

  The literal value's type does not match the input field's declared type.

SEE ALSO

JumpIf, JumpTo, Mark

JumpTo

control | category: control-flow | no `||` | resultSize: 1

Unconditionally jumps to a named `Mark` later in the rule chain.

`JumpTo` skips the remaining rules between itself and the named `Mark`. It accepts no input and produces no output — it must appear as a standalone statement in the chain (no `:field => JumpTo(...)` form).

Typical use: closing the "true" branch of a `JumpIfNot` if/else so control falls through past the "else" branch.

PARAMETERS

1 — `mark` *Mandatory* — accepts: STRING

The name of the destination `Mark` to jump to.

INPUT TYPE

OUTPUT TYPE

No output — `JumpTo` is a standalone control statement.

REQUIRES

- No pipeline input or output is allowed.
- The destination `Mark` must exist in the same chain.

EXAMPLES

Skip the default branch of an if/else

```
:visit_concept => JumpIfNot(9202, "Default") => :visit_concept
// ...processing for the matching case...
JumpTo("Done")
Mark("Default")
// ...default-branch processing...
Mark("Done")
```

ERRORS

- `JumpTo: No named input or output are allowed`

  A `:field => JumpTo(...)` or `JumpTo(...) => :field` form was attempted.

- `JumpTo accepts a single STRING that defines the mark to jump to`

  Wrong number of parameters or non-string parameter.

SEE ALSO

JumpIf, JumpIfNot, Mark

Mark

control | category: control-flow | no `||` | resultSize: 1

Declares a labelled jump target reachable by `JumpIf`, `JumpIfNot`, and `JumpTo`.

`Mark` is a no-op at runtime; it merely records its name as a position in the rule chain. When a `Jump*` rule fires, execution resumes at the matching `Mark`. Marks must be unique per chain, and every jump must reference an existing mark.

PARAMETERS

1 — `name` *Mandatory* — accepts: STRING

A unique label within the rule chain. Other rules in the same chain (`JumpIf`, `JumpIfNot`, `JumpTo`) reference this name.

INPUT TYPE

OUTPUT TYPE

No output — `Mark` is a standalone label statement.

REQUIRES

- No pipeline input or output is allowed.

EXAMPLES

Declare two marks for a two-branch if/else

```
:visit_concept => JumpIfNot(9202, "Default") => :visit_concept
JumpTo("Done")
Mark("Default")
// default-branch logic
Mark("Done")
```

ERRORS

- `Mark: No named input or output are allowed`

  A `:field => Mark(...)` form was attempted. `Mark` is a label only.

- `Mark accepts a single STRING that defines the mark`

  Wrong number of parameters or non-string parameter.

SEE ALSO

JumpIf, JumpIfNot, JumpTo

5.2 Configuration

5.2.1 Configuration reference

The `etl.conf` file ships with the engine and contains a demonstration of every supported block. The configuration consists of six top-level blocks:

- `source database` — where the engine reads from.
- `cdm database` — the destination database hosting the OMOP CDM.
- `rabbit configuration` — locations of OHDSI Rabbit artefacts (Usagi files, the Rabbit-in-a-Hat file, index storage).
- `automation` — `pre` and `post` automation steps.
- `tool configuration` — engine-wide knobs (worker threads, license, slot model).
- `logging` — log level and directory.

Each block is reviewed in turn on the linked pages.

Path resolution

Each path in the configuration file can be either relative or absolute. Relative paths are resolved against the location of `etl.conf` itself. When running the ETL Engine from inside its container, it is recommended to keep the pre-entered paths from the configuration template and folder structure intact.

5.2.2 `source database`

Defines the driver used to read the source data and the connection string for that driver. Valid driver values are:

- `csv`
- `postgresql` (native libpq)
- `psql-odbc` (PostgreSQL over ODBC)
- `mariadb`
- `sqlserver`

The `csv` driver only requires a path to the directory containing the CSV files. The `postgresql` and `mariadb` drivers take a key=value space-separated connection string — see `cdm database` for the full format, which is identical between source and CDM blocks. The `psql-odbc` and `sqlserver` drivers use ODBC connection strings in the semicolon-separated format — see Drivers.

Example

```
source database {
  driver: "csv"
  connection string: "/etc/etl-engine.d/db/"
}
```

If you use the `csv` driver, drop the source CSV files into `etc/etl-engine.d/db/`. Each path may be absolute or relative; relative paths resolve against the location of `etl.conf`.

5.2.3 `cdm database`

Defines the destination database for the OMOP CDM. This database also holds the OMOP vocabulary used to resolve concept IDs from the source data.

Valid driver values: `csv`, `postgresql`, `psql-odbc`, `mariadb`, `sqlserver`. (CSV is mostly useful as a target for one-shot exports; the RDBMS drivers are the typical production choice. `postgresql` is the native libpq path and is preferred over `psql-odbc` unless your environment requires ODBC.)

Connection strings

`mariadb` AND `postgresql`

Both drivers use the same key=value space-separated format:

Key	Meaning
`user`	Username used to connect.
`password`	Password for `user`.
`host`	Resolvable hostname or IP address.
`port`	Connection port.
`dbname`	Database name.
`options`	(PostgreSQL only) e.g. `--search_path=omop_cdm`.

```
cdm database {
  driver: "postgresql"
  connection string: "user=postgres host=<hostname> port=5432
dbname=postgres options='--search_path=omop_cdm'
password=<password>"
}
```

MariaDB has no `options` key — it lacks the schema / search-path concept.

`sqlserver`

The SQL Server driver uses an **ODBC connection string** in the standard semicolon-separated format. The shipped container bundles Microsoft's ODBC Driver 18 for SQL Server.

```
cdm database {
  driver: "sqlserver"
  connection string: "Driver={ODBC Driver 18 for SQL Server};Server=sqlserver.example.com,
1433;Database=omop;UID=etl;PWD=secret;TrustServerCertificate=yes"
}
```

`csv`

The CSV driver writes one file per CDM table:

```
cdm database {
  driver: "csv"
  connection string: "/etc/etl-engine.d/cdm-out/"
}
```

Additional properties

`vocabulary cache size`

How many records the engine caches from the OMOP vocabulary. **Optional**; defaults to **1000**. Increasing it may help on machines with plenty of memory and large mapping workloads.

`max database connections`

Caps the number of concurrent connections the engine opens to the destination database. **Optional**; the engine picks a sensible default based on `worker threads`. Set this lower if the database has a tight connection-limit, or higher if the database can handle more load than the engine is currently giving it.

```
cdm database {
  driver: "postgresql"
  connection string: "..."
  max database connections: 4
}
```

`limit tables`

A block listing CDM tables to include in this run. Only the tables named here are loaded. Useful when debugging a single CDM table, but if a dependent table is excluded from the list, all the dependents that need it will fail to load.

```
cdm database {
  driver: "postgresql"
  connection string: "..."
  limit tables {
    table: person
    table: condition_occurrence
  }
}
```

⚠ **`limit tables` interacts with `clean omop data`**

If `automation.pre.clean omop data` is `true`, only the tables named in `limit tables` are cleaned. This works well when adding new data to independent tables (e.g. `drug_exposure` while `person` is already loaded), but does not work in reverse.

5.2.4 `rabbit configuration`

Points to the artefacts produced by the OHDSI Rabbit family of tools.

Example

```
rabbit configuration {
  usagi directory: "/etc/etl-engine.d/usagi/"
  hat file: "ScanReport.xlsx.json.gz"
}
```

Properties

`usagi directory`

Directory containing the Usagi mapping files referenced by `UsagiMap` rules. **Optional** — if no rule chain calls `UsagiMap`, this key may be omitted.

`hat file`

Path to the Rabbit-in-a-Hat output file containing the source-to-CDM mapping. **Mandatory** for any non-trivial run. Conventionally the hat file lives next to `etl.conf` itself, in `/etc/etl-engine/`.

Related

The `index directory` key — used by `StoreIndexMap` and `StoreSequence` to persist state between runs — lives in `tool configuration`, not here.

5.2.5 `automation`

Toggles the engine's automation steps. Automation runs in two phases:

- `pre` — runs before the mapping starts. Used to set up the destination database (CDM tables, vocabulary, optional cleaning).
- `post` — runs after the mapping completes. Used to populate derived tables (era tables, preceding-visit IDs).

Example

```
automation {
  pre {
    create omop cdm: once
    populate vocabulary: once
    clean omop data: true
    vocabulary directory: /etc/etl-engine.d/vocab/
    vocabulary backup: /etc/etl-engine.d/vocab.backup/
  }

  post {
    add preceding visit ids: true
    create observation periods: true
    create condition eras: true
    create drug eras: true
    create dose eras: true
  }
}
```

`pre` block

`create omop cdm`

Sets up the OMOP CDM tables in the destination CDM — a database **or a CSV output directory** — using the CDM version declared in the Rabbit-in-a-Hat file. On a database target it issues the `CREATE TABLE`s; on a CSV target it writes each non-vocabulary CDM table to its own file with the canonical OMOP header. From there the two behave identically, including when a single CDM table is written by several `etl_map` steps (each appends its columns).

Value	Behaviour
`always`	Build (or rebuild) the CDM on every run.
`once`	Build the CDM only if it does not already exist.
`never`	Do not run this automation. **Default.**

`populate vocabulary`

Populates the vocabulary tables from a zip file downloaded from Athena. The zip must live in `vocabulary directory`.

Value	Behaviour
`always`	Build (or rebuild) the vocabulary on every run.
`once`	Build the vocabulary every time a new zip is present in `vocabulary directory`. After processing, the zip is moved to `vocabulary backup`.
`never`	Do not run this automation. **Default.**

`vocabulary directory`

Directory the engine looks in for the Athena zip. **Default:** `/etc/etl-engine.d/vocab/`.

`vocabulary backup`

Directory the engine moves processed Athena zips into. **Default:** `/etc/etl-engine.d/vocab.backup/`.

`clean omop data`

If `true`, empties all non-vocabulary content from the OMOP CDM before the transformation runs, keeping the table schema. **Optional**; missing values are treated as `false`. On a database target this is a `TRUNCATE`; on a **CSV target** it resets each table file to its canonical OMOP header (empty body) — the same semantics — so a pipeline can clean once and have each subsequent `etl_map` step append to a still-correctly-shaped table.

> **Warning**
>
> Combined with `cdm database.limit tables`, only the listed tables are cleaned. See `cdm database`.

`post` block

All `post` steps default to `false` and may be omitted entirely if not needed.

Key	What it does	CSV target
`create observation periods`	Populates `observation_period` from the span of each person's clinical events.	Yes
`create condition eras`	Populates `condition_era` from `condition_occurrence`.	Yes
`create drug eras`	Populates `drug_era` from `drug_exposure` (rolled up to RxNorm ingredients).	Yes [1]
`create dose eras`	Populates `dose_era` from `drug_exposure` and `drug_strength`.	Yes [1]
`add preceding visit ids`	Populates the preceding-visit IDs in `visit_occurrence` and `visit_detail`.	Yes

> **Derived tables on a CSV target**
>
> `observation_period`, `condition_era`, `drug_era` and `dose_era` are built by reading the produced CDM tables back and folding them in memory, so they work on a **CSV (file) target** as well as a database. The whole derived table is rebuilt from current CDM state on every run, so it stays correct under selective re-runs.
>
> `add preceding visit ids` assigns each visit's predecessor per person (ordered by start datetime/date). On a database target it updates the rows in place; on a CSV target it rewrites the produced `visit_occurrence` / `visit_detail` files in place after they are written. Either way only NULL preceding-id cells are filled, so it is safe under selective re-runs.
>
> [1] On a CSV target the vocabulary is never loaded into the CDM, so `drug_era` and `dose_era` read the Athena vocabulary CSVs directly: `drug_era` needs `CONCEPT.csv` and `CONCEPT_ANCESTOR.csv`, `dose_era` needs `DRUG_STRENGTH.csv`. The engine looks for them in `vocabulary directory` first, then in the CDM output directory. (On a database target these come from the loaded vocabulary instead.)

5.2.6 `tool configuration`

Engine-wide configuration: parallelism, caches, index storage, and any optional execution-mode switches.

Example

```
tool configuration {
  worker threads: 10
  use dedup cache: false
  index directory: "/etc/etl-engine.d/indexes/"
}
```

Properties

`worker threads`

Number of worker threads used to process source rows in parallel. **Optional**; the shipped configuration sets it to `10`. Tune this to the source-data volume and the number of CPU cores available — see Performance Tuning.

`index directory`

Directory used to store and load index data and global sequences across runs. This is what makes it possible to split an ETL across multiple Rabbit-in-a-Hat files: the `StoreIndexMap` and `StoreSequence` table rules persist their state here.

`use dedup cache`

When `true`, caches deduplication results across rows. **Optional**; defaults to `false`. Enable this when many rows resolve to the same `Normalize` key — the cache turns a key re-check into a single hash lookup.

`slot model`

When `"true"`, enables the slot-model executor. **Optional**; defaults to `"false"`. The slot model is the recommended path for new deployments — it pre-resolves field-to-input mappings at compile time and removes several runtime locks. The legacy path remains for compatibility with rules that have not yet been migrated.

`license file`

Path to the engine's license file. **Optional in production deployments** — the shipped Docker image bakes in a default license-file location, so ordinary users do not set this key. Override it only if you are running the engine outside the shipped container or with a non-default license location.

5.2.7 `logging`

Configures where the engine writes its logs and how chatty it is.

Example

```
logging {
  log level: "INFO"
  log directory: "/var/log/etl-engine/"
}
```

`log level`

Five levels are available, from most verbose to least:

Level	What it covers
`TRACE`	Reading the Rabbit-in-a-Hat file, building the internal mapping representation, optimisation choices.
`DEBUG`	Less detail than `TRACE` but still enough to pinpoint where an error originates.
`INFO`	Per-stage messages: loading the Rabbit-in-a-Hat file, cleaning, analysing tables, mapping.
`WARN`	Unexpected behaviour the engine recovered from — e.g. a rule chain that failed to parse.
`ERROR`	Fatal errors only. The engine prints a single line stating the cause and exits.

The `log level` setting **only applies to** `etl-engine.log`. Per-table log files always log at INFO level (this is not configurable).

`log directory`

Directory where the engine writes its log files. When running inside the shipped Docker container, leave this at the default path so the engine's volume mounts work correctly.

See Troubleshooting → Log Files for what each log file contains.

5.3 Glossary

5.3.1 OMOP / OHDSI

OMOP CDM

Observational Medical Outcomes Partnership Common Data Model — a standardised schema for observational health data, maintained by OHDSI.

OHDSI

Observational Health Data Sciences and Informatics — the open-science community that maintains OMOP CDM and the surrounding tooling.

Athena

The OHDSI vocabulary download portal at athena.ohdsi.org.

Rabbit-in-a-Hat

An OHDSI tool for designing source-to-CDM mappings visually. The ETL Engine consumes its JSON output directly.

Usagi

An OHDSI tool for mapping source codes to OMOP standard concepts. The engine reads its CSV output via the `UsagiMap` rule.

WhiteRabbit

An OHDSI tool that produces the scan reports consumed by Rabbit-in-a-Hat.

5.3.2 DSL

Rule chain

A sequence of rules executed top-down on a single field, field-to-field mapping, or CDM-table input.

Mapping rule

A rule that transforms one input value into one output value. Most rules of the form `AsX`, `XFromDate`, etc. are mapping rules.

Aggregating rule

A rule that takes any number of inputs and produces exactly one output. `Add`, `Min`, `Use`, `Selector`, etc. are aggregating rules.

Table rule

A rule placed on a CDM table itself (rather than on a field), affecting how rows enter the table. `DependOn`, `Normalize`, `Join` are examples.

Filter rule

A rule that decides which rows pass through, based on a grouping key. `First` and `Last` are the only filter rules currently.

Virtual field

A field that exists only inside the engine, not in the CDM database. Declared with `field:` in a CDM-table Comments field. See Virtual Fields.

Multi-sized result

A rule output that produces more than one record per input row. Marked with the `[]` or `[N]` suffix on the rule.

Conditional chaining

The `||` operator joining rules into a fallback sequence: try the first, fall through to the next on failure. See Conditional Chaining.

5.4 Third-party licenses

This page covers the licensing of everything in the engine's ecosystem that is *not* the engine itself:

- Demo data — the Synthmas synthetic patient dataset that `bin/setup-synthmas.sh` fetches.
- Reference vocabularies — the OMOP vocabulary downloaded from Athena.
- Open-source libraries — every third-party library the engine bundles inside its Docker image, with full licence text reproduced as required.

For the **ETL Engine's own commercial licensing** — Free / Paid tiers, support, and contact — see Licensing.

5.4.1 Synthmas (synthetic patient dataset)

Generated by Synthea, MITRE Corporation, **Apache 2.0**. The Synthmas dataset MITRE publishes is for unrestricted research and educational use. No real-patient data is involved.

`bin/setup-synthmas.sh` pulls Synthmas from MITRE's canonical URL — Rook IT does not redistribute the dataset.

5.4.2 OMOP vocabularies (Athena)

The OMOP standard vocabulary distributed by Athena is an aggregation of many independently licensed vocabularies. Each download bundles whichever subset you selected, governed by the licences in effect for those subsets — including (non-exhaustive):

- **UMLS Metathesaurus** — requires a UMLS Metathesaurus Licence Agreement (free for most uses; acceptance required).
- **SNOMED CT** — IHTSDO terms; free for member countries, licensed elsewhere.
- **CPT-4** — AMA-licensed; requires an AMA agreement and the separate unpacking-script step described in Troubleshooting → Athena zips with licensed code sets.
- **LOINC** — Regenstrief Institute, free with attribution.
- **RxNorm** — National Library of Medicine, free.
- **ICD-10-CM**, **ICD-9-CM** — public domain in most jurisdictions.

By downloading vocabularies from Athena you accept the terms each constituent vocabulary requires. **Rook IT does not redistribute the vocabulary** — `bin/setup-vocab.sh` only operates on a zip you have already downloaded yourself.

5.4.3 Open-source libraries bundled in the engine

The libraries below ship inside the engine's Docker image. Their licence texts are reproduced in full further down this page and are also included verbatim under `licenses/` in the unpacked engine artefact.

Library	What it does for the engine	License
Boost	JSON parser and the engine's value/object model.	Boost Software License 1.0
ConfigTool	Generates the parser for `etl.conf`.	Beerware License (Revision 42)
csv2	CSV input parser for source files and Usagi maps; CSV writer.	MIT
fmt	String formatting in log output and elsewhere.	MIT
MariaDB Connector/C	MariaDB / MySQL driver.	LGPL 2.1
MemoryPool	Object pool used in the engine's hot-path allocators.	MIT
minizip	Reads Athena vocabulary zip archives during automation.	zlib (with attribution)
mio	Memory-mapped file I/O for fast index loading.	MIT
PCRE2	Regular-expression engine for `Regex` and `Map` regex keys.	BSD 3-Clause
PostgreSQL libpq	PostgreSQL driver.	PostgreSQL License (BSD-style)
SOCI	Database-abstraction layer used by the RDBMS drivers.	Boost Software License 1.0
spdlog	Logging framework underlying the engine's per-table log files.	MIT
zlib	Decompression of `.gz` Rabbit-in-a-Hat files and Athena zips.	zlib

5.4.4 Boost

The engine uses Boost for its JSON parser and the value/object model that flows through the transformation pipeline.

> **Boost Software License 1.0 — full text**

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

5.4.5 ConfigTool

ConfigTool generates the parser used to read the `etl.conf` configuration file.

Beerware License (Revision 42) — full text

```
----------------------------------------------------------------------------
"THE BEER-WARE LICENSE" (Revision 42):
<carsten@stiborg.dk> wrote this file. As long as you retain this notice you
can do whatever you want with this stuff. If we meet some day, and you think
this stuff is worth it, you can buy me a beer in return. Carsten Stiborg
----------------------------------------------------------------------------
```

5.4.6 csv2

csv2 is the CSV reader for source files (when the source driver is `csv`) and Usagi mapping files. It is also the writer when CSV is used as a CDM target.

MIT License — full text

```
MIT License

Copyright (c) 2019 Pranav

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

5.4.7 fmt

fmt formats log output and a number of internal string-building call sites.

MIT License — full text

```
Copyright (c) 2012 - present, Victor Zverovich

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

--- Optional exception to the license ---

As an exception, if, as a result of your compiling your source code, portions of this Software are embedded into a machine-executable object form of such
source code, you may redistribute such embedded portions in such object form without including the above copyright and permission notices.
```

5.4.8 MariaDB Connector/C

The MariaDB Connector/C is the wire-protocol driver used for MariaDB and MySQL sources and CDMs.

LGPL 2.1 — full text

```
GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom
to share and change it. By contrast, the GNU General Public Licenses
are intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it. You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations
below.

When we speak of free software, we are referring to freedom of use,
not price. Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights. These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you. You must make sure that they, too, receive or can get the source
code. If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there
is no warranty for the free library. Also, if the library is modified
by someone else and passed on, the recipients should know that what
they have is not the original version, so that the original author's
reputation will not be affected by problems that might be introduced
by others.

Finally, software patents pose a constant threat to the existence of
any free program. We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder. Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License. This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License. We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

When a program is linked with a library, whether statically or using a
shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library. The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom. The Lesser General
Public License permits more lax criteria for linking other code with
the library.

We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
```

```
Public License. It also provides other free software developers Less
of an advantage over competing non-free programs. These disadvantages
are the reason we use the ordinary General Public License for many
libraries. However, the Lesser license provides advantages in certain
special circumstances.

For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it
becomes a de-facto standard. To achieve this, non-free programs must
be allowed to use the library. A more frequent case is that a free
library does the same job as widely used non-free libraries. In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software. For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and
modification follow. Pay close attention to the difference between a
"work based on the library" and a "work that uses the library". The
former contains code derived from the library, whereas the latter must
be combined with the library in order to run.  TERMS AND CONDITIONS
FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License"). Each
licensee is addressed as "you".

A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work
which has been distributed under these terms. A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language. (Hereinafter, translation is
included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for
making modifications to it. For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control
compilation and installation of the library.

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope. The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it). Whether that is true depends on what the Library does and
what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a
fee.

2. You may modify your copy or copies of the Library or any portion of
it, thus forming a work based on the Library, and copy and distribute
such modifications or work under the terms of Section 1 above,
provided that you also meet all of these conditions:

    * a) The modified work must itself be a software library.

    * b) You must cause the files modified to carry prominent notices
      stating that you changed the files and the date of any change.

    * c) You must cause the whole of the work to be licensed at no
        charge to all third parties under the terms of this License.
```

```
    * d) If a facility in the modified Library refers to a function or
      a table of data to be supplied by an application program that
      uses the facility, other than as an argument passed when the
      facility is invoked, then you must make a good faith effort to
      ensure that, in the event an application does not supply such
      function or table, the facility still operates, and performs
      whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots
      has a purpose that is entirely well-defined independent of the
      application. Therefore, Subsection 2d requires that any
      application-supplied function or table used by this function
      must be optional: if the application does not supply it, the
      square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library. To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License. (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.) Do not make any other change in
these notices.

Once this change is made in a given copy, it is irreversible for that
copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the
Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative
of it, under Section 2) in object code or executable form under the
terms of Sections 1 and 2 above provided that you accompany it with
the complete corresponding machine-readable source code, which must be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange.

If distribution of object code is made by offering access to copy from
a designated place, then offering equivalent access to copy the source
code from the same place satisfies the requirement to distribute the
source code, even though third parties are not compelled to copy the
source along with the object code.

5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library". Such a work,
in isolation, is not a derivative work of the Library, and therefore
falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library". The executable is therefore covered by this License. Section
6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is
not. Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library. The
threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure
layouts and accessors, and small macros and small inline functions
(ten lines or less in length), then the use of the object file is
unrestricted, regardless of whether it is legally a derivative
```

work. (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section
6. Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link
a "work that uses the Library" with the Library to produce a work
containing portions of the Library, and distribute that work under
terms of your choice, provided that the terms permit modification of
the work for the customer's own use and reverse engineering for
debugging such modifications.

You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License. You must supply a copy of this License. If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License. Also, you must do one
of these things:

    * a) Accompany the work with the complete corresponding
      machine-readable source code for the Library including whatever
      changes were used in the work (which must be distributed under
      Sections 1 and 2 above); and, if the work is an executable
      linked with the Library, with the complete machine-readable
      "work that uses the Library", as object code and/or source code,
      so that the user can modify the Library and then relink to
      produce a modified executable containing the modified
      Library. (It is understood that the user who changes the
      contents of definitions files in the Library will not
      necessarily be able to recompile the application to use the
      modified definitions.)
    * b) Use a suitable shared library mechanism for linking with the
      Library. A suitable mechanism is one that (1) uses at run time a
      copy of the library already present on the user's computer
      system, rather than copying library functions into the
      executable, and (2) will operate properly with a modified
      version of the library, if the user installs one, as long as the
      modified version is interface-compatible with the version that
      the work was made with.
    * c) Accompany the work with a written offer, valid for at least
      three years, to give the same user the materials specified in
      Subsection 6a, above, for a charge no more than the cost of
      performing this distribution.
    * d) If distribution of the work is made by offering access to
      copy from a designated place, offer equivalent access to copy
      the above specified materials from the same place.
    * e) Verify that the user has already received a copy of these
      materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it. However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
the executable.

It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system. Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.

7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:

    * a) Accompany the combined library with a copy of the same work
      based on the Library, uncombined with any other library
      facilities. This must be distributed under the terms of the
      Sections above.

    * b) Give prominent notice with the combined library of the fact
      that part of it is a work based on the Library, and explaining
      where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the
Library except as expressly provided under this License. Any attempt
otherwise to copy, modify, sublicense, link with, or distribute the
Library is void, and will automatically terminate your rights under
this License. However, parties who have received copies, or rights,

```
from you under this License will not have their licenses terminated so
long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Library or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted
herein. You are not responsible for enforcing compliance by third
parties with this License.

11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all. For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply, and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time. Such
new versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation. If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission. For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this. Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE
LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS
AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF
ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
```

```
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.  END OF TERMS AND CONDITIONS.
```

5.4.9 MemoryPool

MemoryPool is the small-object allocator used on the value-allocation hot path.

MIT License — full text

```
This code is distributed under the MIT License, which is reproduced below and at the top of the project files. This pretty much means you can do whatever you
want with the code, but I will not be liable for ANY kind of damage that this code might cause. Here is the full license which you should read before using
the code:

Copyright (c) 2013 Cosku Acay, http://www.coskuacay.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

5.4.10 minizip

minizip is used to read the Athena vocabulary zip archive during the `populate vocabulary` automation step.

minizip license (zlib-style with attribution) — full text

```
minizip - Copyright (c) 2012 - by Daniel Eggert

iOS / Cocoa version of minizip

Modified to work on iOS (while not caring about anything else). Wraps the archive generation into a class.




MiniZip - Copyright (c) 1998-2010 - by Gilles Vollant - version 1.1 64 bits from Mathias Svensson

Introduction
---------------------
MiniZip 1.1 is built from MiniZip 1.0 by Gilles Vollant ( http://www.winimage.com/zLibDll/minizip.html )

When adding ZIP64 support into minizip it would result into risk of breaking compatibility with minizip 1.0.
All possible work was done for compatibility.


Background
---------------------
When adding ZIP64 support Mathias Svensson found that Even Rouault have added ZIP64
support for unzip.c into minizip for a open source project called gdal ( http://www.gdal.org/ )

That was used as a starting point. And after that ZIP64 support was added to zip.c
some refactoring and code cleanup was also done.


Changed from MiniZip 1.0 to MiniZip 1.1
---------------------------------------
* Added ZIP64 support for unzip ( by Even Rouault )
* Added ZIP64 support for zip ( by Mathias Svensson )
* Reverted some changed that Even Rouault did.
* Bunch of patches received from Gulles Vollant that he received for MiniZip from various users.
* Added unzip patch for BZIP Compression method (patch create by Daniel Borca)
* Added BZIP Compress method for zip
* Did some refactoring and code cleanup


Credits

 Gilles Vollant     - Original MiniZip author
 Even Rouault       - ZIP64 unzip Support
 Daniel Borca       - BZip Compression method support in unzip
 Mathias Svensson   - ZIP64 zip support
 Mathias Svensson   - BZip Compression method support in zip

 Resources

 ZipLayout   http://result42.com/projects/ZipFileLayout
             Command line tool for Windows that shows the layout and information of the headers in a zip archive.
             Used when debugging and validating the creation of zip files using MiniZip64


 ZIP App Note  http://www.pkware.com/documents/casestudies/APPNOTE.TXT
               Zip File specification


Notes.
 * To be able to use BZip compression method in zip64.c or unzip64.c the BZIP2 lib is needed and HAVE_BZIP2 need to be defined.

License
------------------------------------------------------------
   Condition of use and distribution are the same than zlib :

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose,
  including commercial applications, and to alter it and redistribute it
  freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
     claim that you wrote the original software. If you use this software
     in a product, an acknowledgment in the product documentation would be
     appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be
     misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.
```

```
------------------------------------------------------------
```

5.4.11 mio

mio provides memory-mapped file I/O, used by the index-store loader for fast startup when an ETL is split across multiple Rabbit-in-a-Hat files.

MIT License — full text

```
MIT License

Copyright (c) 2018 https://github.com/mandreyel/

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

5.4.12 PCRE2

PCRE2 is the regular-expression engine that backs the `Regex` rule and the regex keys accepted by the `Map` rule.

BSD 3-Clause — full text

```
PCRE2 LICENCE
-------------

PCRE2 is a library of functions to support regular expressions whose syntax
and semantics are as close as possible to those of the Perl 5 language.

Releases 10.00 and above of PCRE2 are distributed under the terms of the "BSD"
licence, as specified below, with one exemption for certain binary
redistributions. The documentation for PCRE2, supplied in the "doc" directory,
is distributed under the same terms as the software itself. The data in the
testdata directory is not copyrighted and is in the public domain.

The basic library functions are written in C and are freestanding. Also
included in the distribution is a just-in-time compiler that can be used to
optimize pattern matching. This is an optional feature that can be omitted when
the library is built.


THE BASIC LIBRARY FUNCTIONS
---------------------------

Written by:       Philip Hazel
Email local part: Philip.Hazel
Email domain:     gmail.com

Retired from University of Cambridge Computing Service,
Cambridge, England.

Copyright (c) 1997-2021 University of Cambridge
All rights reserved.


PCRE2 JUST-IN-TIME COMPILATION SUPPORT
--------------------------------------

Written by:       Zoltan Herczeg
Email local part: hzmester
Email domain:     freemail.hu

Copyright(c) 2010-2021 Zoltan Herczeg
All rights reserved.


STACK-LESS JUST-IN-TIME COMPILER
--------------------------------

Written by:       Zoltan Herczeg
Email local part: hzmester
Email domain:     freemail.hu

Copyright(c) 2009-2021 Zoltan Herczeg
All rights reserved.


THE "BSD" LICENCE
-----------------

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notices,
      this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above copyright
      notices, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.

    * Neither the name of the University of Cambridge nor the names of any
      contributors may be used to endorse or promote products derived from this
      software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

```
EXEMPTION FOR BINARY LIBRARY-LIKE PACKAGES
------------------------------------------

The second condition in the BSD licence (covering binary redistributions) does
not apply all the way down a chain of software. If binary package A includes
PCRE2, it must respect the condition, but if package B is software that
includes package A, the condition is not imposed on package B unless it uses
PCRE2 independently.

End
```

5.4.13 PostgreSQL libpq

The PostgreSQL `libpq` client library is the wire-protocol driver used for PostgreSQL sources and CDMs.

> **PostgreSQL License (BSD-style) — full text**

```
PostgreSQL is released under the PostgreSQL License, a liberal Open Source license, similar to the BSD or MIT licenses.

PostgreSQL Database Management System
(formerly known as Postgres, then as Postgres95)

Portions Copyright © 1996-2023, The PostgreSQL Global Development Group

Portions Copyright © 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby
granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST
PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE,
SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
```

5.4.14 SOCI

SOCI is the database-abstraction layer that all the RDBMS drivers (MariaDB, PostgreSQL, SQL Server) sit on top of.

> **Boost Software License 1.0 — full text**

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

5.4.15 spdlog

spdlog is the logging framework that produces the engine's `etl-engine.log` and the per-table log files under `logs/`.

MIT License — full text

```
The MIT License (MIT)

Copyright (c) 2016 Gabi Melman.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

-- NOTE: Third party dependency used by this software --
This software depends on the fmt lib (MIT License),
and users must comply to its license: https://github.com/fmtlib/fmt/blob/master/LICENSE.rst
```

5.4.16 zlib

zlib decompresses `.gz` files at the input boundary — both the Rabbit-in-a-Hat hat file (which is gzipped JSON) and Athena vocabulary zip archives.

zlib License — full text

```
/* zlib.h -- interface of the 'zlib' general purpose compression library
  version 1.3, August 18th, 2022

  Copyright (C) 1995-2023 Jean-loup Gailly and Mark Adler

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose,
  including commercial applications, and to alter it and redistribute it
  freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
     claim that you wrote the original software. If you use this software
     in a product, an acknowledgment in the product documentation would be
     appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be
     misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.

  Jean-loup Gailly        Mark Adler
  jloup@gzip.org          madler@alumni.caltech.edu

*/
```