UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Siim Neljandik

Understanding and Visualizing Data Lineage in Health Data Transformation Pipelines

Bachelor's Thesis (9 ECTS)

Supervisor:
Harry-Anton Talvik, MSc

Tartu 2025

Understanding and Visualizing Data Lineage in Health Data Transformation Pipelines

Abstract:

As modern data transformation pipelines grow in complexity and scale, identifying the origins of data becomes increasingly difficult. This is especially true for health data, where its sensitivity and potential individual impact make it essential to trace its originators without revealing personal information in the final product. Due to strict international regulations such as the *Health Insurance Portability and Accountability Act* and the *General Data Protection Regulation*, understanding data lineage is important for compliance.

More crucially, in transformation pipelines, data lineage enables users to trace data sources from the final output and assess the impact of data or system changes on the end result. This thesis reviews currently available tools and their underlying parsing libraries. We identified tools suitable for an example health data transformation pipeline and from them deployed Tokern, Marquez, and SQLLineage. During deployment, we examined their limitations and extended them where feasible to enable full functionality.

Keywords: Data Lineage, Database, SQL

CERCS: P175 Informatics, systems theory

Andmevoogude mõistmine ja visualiseerimine terviseandmete töötlemise protsessides

Lühikokkuvõte:

Kaasaegsed andmete teisenduskonveierid muutuvad üha keerukamaks ja mastaapsemaks ja andmete päritolu tuvastamine järjest keerukamaks. See kehtib eriti terviseandmete kohta, mille tundlikkuse ja võimaliku individuaalse mõju tõttu on oluline tuvastada andmete päritolu, ilma isikuandmeid avalikustamata.

Andmevoogude mõistmine on oluline ka rangete rahvusvaheliste regulatsioonidega (näiteks USA *Health Insurance Portability and Accountability Act* ja Euroopa Liidu Isikuandmete Kaitse Üldmäärus) kooskõlas olemiseks.

Ülioluline on ka teisenduskonveieri andmete päritolu mõistmine. See võimaldab kasutajatel leida lõppväljundist andmete algallikad ning hinnata andmete ja süsteemi muutuste mõju lõpptulemusele. Käesolevas lõputöös uurisime olemasolevaid tööriistu ning nende poolt kasutatud süntaksianalüüsi teeke. Töö tulemusena tuvastasime tööriistad, mis sobivad praktilises näitena käsitletud terviseandmete töötlemise protsessi jaoks. Nendest osutusid väljavalituks Tokern, Marquez ja SQLLineage, mille paigutasime teisenduskonveierile. Paigutuse käigus uurisime nende tööriistade piiranguid ja laiendasime neid kus vajalik, et saavutada soovitud funktsionaalsus.

Võtmesõnad: Andmevood, SQL, andmebaas

CERCS: P175 Informaatika, süsteemiteooria

Contents

1. Introduction 6
  1.1 Problem Statement 6
  1.2 Contributions 6
  1.3 Road Map 6
  1.4 Use of Generative AI 7
  1.5 Funding 7
2. Introduction to Data Lineage 8
  2.1 What is Data Lineage 8
  2.2 Taxonomy 8
  2.3 *Design Lineage* and *Operational Lineage* 8
  2.4 *Coarse-Grain* Lineage and *Fine-Grain* Lineage 9
  2.5 *Where-*, *Why-*, and *How*-Lineage 10
3. Comparison of Existing Data Lineage Tools 11
  3.1 Introduction to Lineage Tools 11
  3.2 The Comparison of Lineage Tools 12
  3.3 Summary of Results 12
4. Underlying SQL Processing Libraries and Usage 14
  4.1 SQLFluff 14
  4.2 SQLParse 15
  4.3 SQLGlot 16
  4.4 OpenLineage Parser 17
  4.5 Comparison of Libraries 18
5. Practical Health Data Transformation Example 19
  5.1 Description of the Example 19
    5.1.1 Database 19
    5.1.2 Orchestration Pipeline Code 20
  5.2 Tokern 21
    5.2.1 Initial Setup Process 21
    5.2.2 Usage of Tools 22
  5.3 OpenLineage and Marquez 22
    5.3.1 Initial Setup Process 22
    5.3.2 Usage of Tools 22

5.3.3 Extending the Tools 24
5.3.4 Final Lineage Output 24
5.4 SQLLineage 25
5.4.1 Initial Setup Process 25
5.4.2 Usage of Tools 25
5.4.3 Extending the Tools 26
5.4.4 Final Lineage Output 28
5.5 Summary Deployment Results 28
6. Conclusion 33
References 34
Appendices 36
I. Glossary 36
II. SQL Transformation Script 37
III. OpenLineage Job Start Call 41
IV. OpenLineage Job Completion Call 43
License 46

1. Introduction

As data projects evolve over time, comprehending their complete data flow and relationships becomes increasingly complex. This challenge is particularly pronounced in projects with changing contributors, where team turnover can lead to a gradual loss of institutional knowledge and, consequently, obscure critical information regarding *data lineage* and *provenance*.

1.1 Problem Statement

The primary objective of this research is to investigate current standards, methodologies, and tools related to data lineage. Based on these findings, the study aims to incorporate or adapt existing tools to produce a high-level representation of data lineage for the specified transformations.

1.2 Contributions

This thesis aims to provide an overview of data lineage tools and libraries suitable for use in a health data transformation pipeline. In addition, after identifying appropriate tools for deployment on the example pipeline, we implement and evaluate lineage tools by generating example outputs for final review. The authors' most significant practical contribution lies in deploying these tools and extending them where necessary to align with the example pipeline.

1.3 Road Map

The thesis is composed as follows:

1. Chapter 2 (Introduction to Data Lineage) introduces data lineage and outlines a taxonomy that can be used to compare different lineage tools;
2. Chapter 3 (Comparison of Existing Data Lineage Tools) reviews various approaches for comparing lineage tools and evaluates a selection of existing tools for potential use in the practical example;
3. Chapter 4 (Underlying SQL Processing Libraries and Usage) examines four libraries used by selected tools for *Structured Query Language* (SQL) parsing and related capabilities;
4. Chapter 5 (Practical Health Data Transformation Example) applies the findings from Chapters 3 and 4 to a representative health data transformation pipeline, assessing which tools are best suited for generating data lineage, either out-of-the-box or with extensions;
5. Chapter 6 (Conclusion) summarizes the findings of the tool review and discusses directions for future research.

In addition, we have the following appendices:

1. Appendix I (Glossary) for terms and definitions;
2. Appendix II (SQL Transformation Script) contains the SQL code that was used in the Bash and SQL-based pipeline for data transformations;
3. Appendix III (OpenLineage Job Start Call) contains the script that was used to make the transformation job start Application Programming Interface (API) call for OpenLineage;
4. Appendix IV (OpenLineage Job Completion Call) contains the script that was used to make the transformation job completion API call for OpenLineage.

1.4 Use of Generative AI

OpenAI GPT-4o and Grammarly were used in this thesis as supporting tools to critique existing text and improve the wording and formatting. All text was subsequently reviewed and edited by the author, who takes full responsibility for the final content.

1.5 Funding

This work was supported in part by the Estonian Research Council through project PRG1844 "Discovery and Analysis of Clinical Pathways in Health Data" (01 Jan 2023–31 Dec 2027), which supplied the data-processing pipeline whose relevant portion served as the main practical example in this study.

2. Introduction to Data Lineage

2.1 What is Data Lineage

Data lineage, sometimes called data provenance, refers to tracking the source and transformation of data within an information system [1]. Data lineage is necessary to manage data effectively and to understand its origin, authenticity, integrity, and trustworthiness. Understanding this will simplify working within these information systems.

Data lineage is especially essential in health *data pipelines* as comprehending the root cause of data changes can directly impact patient outcomes. Additionally, health data involves highly sensitive personal information. Therefore, clearly understanding what data is exposed, and how and where it is shared, is crucial to avoiding violations of privacy regulations such as *Health Insurance Portability and Accountability Act (HIPAA)* [2] or *General Data Protection Regulation (GDPR)* [3]. Violating these regulations can be costly. For example, in 2018, Anthem Inc. was required to pay a $16 million dollar settlement due to a health data breach [4].

2.2 Taxonomy

Data lineage can be categorized in multiple ways. Lineage can be divided into *design lineage* and *operational lineage* [5] or into the *fine grain* (also known as *dataflow*) and *coarse grain* (also known as *workflow*) lineage [6]. *Fine-grain* lineage itself is further categorized into *why-provenance, where-provenance,* and *how-provenance* (see Figure 1) [1, 7].

2.3 *Design Lineage* and *Operational Lineage*

Design lineage describes all digital resources and their linkages. It can be generated using *Extract, Transform, Load (ETL)* tools, captured within the *DevOps* pipeline, or created through automatic cataloging of digital resources as they are added to the environment. *Design lineage* can be further specified as *business lineage*, a simplified view intended for business users to audit data processing activities. For example, *business lineage* may focus solely on the transformation steps within the system.

Operational lineage is achieved by supplementing *design lineage* with metadata produced by data processing engines during data operations. It enables organizations to validate their processes and primarily focuses on capturing the dynamic aspects of lineage [5].

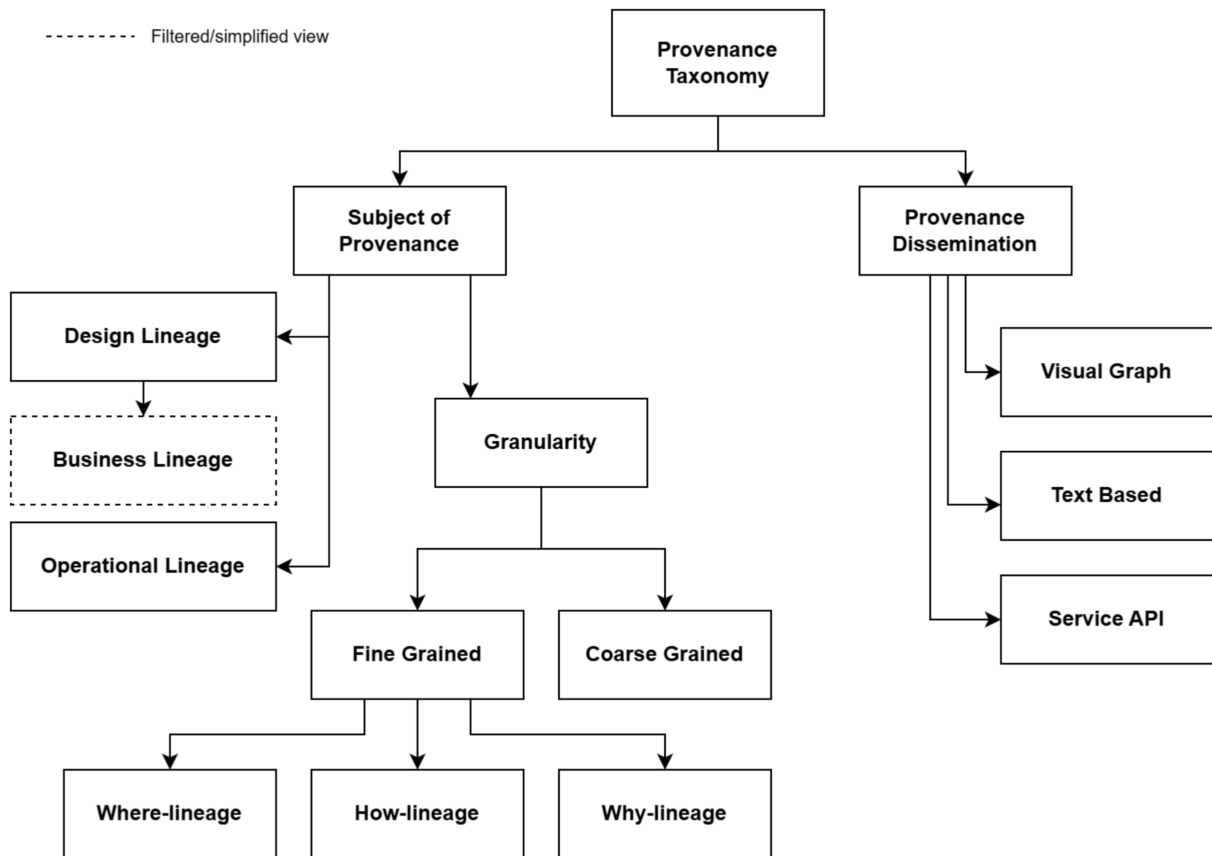


Figure 1. Data provenance taxonomy. Adapted from ”A Survey of Data Provenance in e-Science”, by Y. Simmhan, B. Plale, and D. Gannon, SIGMOD Record, 2005,p 33.

2.4 *Coarse-Grain* Lineage and *Fine-Grain* Lineage

Coarse-grain data lineage records the complete history of the derivation of a dataset, including interactions with humans, software systems, and external devices. This type of lineage emphasizes high-level tracking of the systems and environments through which the dataset passes. In the context of health information, it may begin with the physical collection of data from a patient, followed by its digitization, and subsequent transitions through various information systems until reaching the final output.

Fine-grain data lineage, in contrast, describes the detailed movement of data through a network of databases, with particular attention to specific components and transformations. It focuses on the internal digital journey of the data, capturing granular operations and modifications at each step. For example, in a health data pipeline, it would track changes applied to the digital record at every stage, even within the same process or system [6].

2.5 *Where-*, *Why-*, and *How-*Lineage

Where-, *why*-, and *how*-lineage each cover a portion of full data lineage:

- *Where*-lineage identifies the source elements from which the data was extracted;
- *Why*-lineage provides justification for why data elements appear in the output;
- *How*-lineage describes how parts of the input influence specific parts of the output;

Consider the following SQL query:

```sql
SELECT
    diag.epi_id as document_id,
    UPPER(pat.pat_id) AS patient_id,
    diag.compl_diag_name AS diagnosis
FROM raw.complication_diagnosis AS diag
LEFT JOIN raw.patient AS pat
    ON diag.epi_id = pat.epi_id;
```

The *where*-lineage of the `patient_id` column identifies the original table and column. In the example SQL query, the `patient_id` column originates from the `pat_id` column of the `raw.patient` table.

The *why*-lineage of the `patient_id` column describes the conditions found in the JOIN or WHERE clauses of the SQL query. In the example, the condition is that the `epi_id` column in the `raw.patient` table must match the `epi_id` column in the `raw.complication_diagnosis` table.

The *how*-lineage describes the data transformation logic applied to the `patient_id` column. In the example, it shows the use of the UPPER() function, which converts the `pat.pat_id` column into uppercase characters.

3. Comparison of Existing Data Lineage Tools

3.1 Introduction to Lineage Tools

The need for data lineage has led to the creation of numerous different tools. These tools can be complex, costly, and opaque, making the selection of an appropriate option important. When choosing a tool, several considerations must be taken into account to ensure it matches the intended use case. First and foremost, it is essential to understand the type of lineage required and achievable. For instance, if the goal is simply to identify source elements, a tool that provides only *Where*-lineage might suffice. However, if detailed information about all performed taken on the data is needed, a more comprehensive tool offering *How*-, *Where*-, and *Why*-lineage should be selected.

Additionally, several other factors will influence the decision:

1. Supported technology - The selected tool should support one's technology stack; for example, it should understand the syntax of the chosen database system;
2. Free and open-source versus closed-source and paid software - Closed-source, paid software often includes customer support, assisting an organization with tool implementation, and addressing any issues during use. However, this typically comes at a high cost that might not be feasible. A viable alternative can be free, open-source software, which allows anyone to review, modify, or extend the source code as needed. Nevertheless, open-source solutions often require higher technical expertise compared to commercial off-the-shelf software. Moreover, open-source projects can sometimes be abandoned, halting development and support for the tool;
3. Self-hosted versus *Software-as-a-Service* - Many commercial tools are offered as a service, hosted by the vendor. This can simplify deployment and maintenance, but requires granting external access to internal data or data structures. Self-hosted tools ensure that all data remains on-premises, which can be critical when handling sensitive information;
4. Automation - Some lineage tools are capable of automatically processing *data pipelines* and generating lineage information. Even partial automation can simplify deployment and ensure lineage stays up-to-date. This feature is particularly beneficial for codebases managed by many contributors, such as university projects;

5. AI and LLM features - With the recent mainstream emergence of *large language models* (*LLMs*), several products now claim to incorporate *artificial intelligence (AI)* technologies. This capability is considered valuable by many organizations;
6. Provenance dissemination - Most tools offer a derivation graph to represent data lineage, while others may only provide a simple text-based relationship. Visual representations can be static, annotatable, or modifiable after generation.

In our current reviews, we will not focus on the *AI* and *LLM* features, but all other factors are taken into account in our tool review and selection.

3.2 The Comparison of Lineage Tools

The tools were found by searching previous review articles on Google Scholar by searching for ”data lineage tools” (e.g., [8]). and by reviewing articles found by searching Google for ”open source SQL lineage tools” (e.g., [9]).

In our tool selection, we focused on free, self-hosted, and open source tools that provide *Where*- and *How*-lineage. In addition, we preferred practical lineage-first tools to transformation tools that also offer lineage. This is due to the fact that our transformation pipeline is based on custom code. From the initial review, we identified and compared General SQL Parser, SQLLineage, Tokern, OpenMetadata, and OpenLineage+Marquez (see Table 1).

After mapping the tools and reviewing their comparative strengths and weaknesses, we needed to select tools for our practical example.

3.3 Summary of Results

Based on our requirements, we selected three tools for further evaluation:

1. SQLLineage: A simple, Python-based SQL parser that analyzes queries or SQL script files and returns lineage data. It visualizes results through an Apache web server interface. This tool was chosen for its simplicity and suitability for SQL-centric data transformation systems;
2. Tokern: Chosen for its potential to enable automated lineage creation across the full Python and SQL technology stack;

Data Lineage Tool	Lineage type	Licensing	Latest Release	Supported Query Automation	Supported Code Automation
General SQL Parser [10]	Where and How	Commercial	2024-11-02	PostgreSQL, MySQL +12 more	C#, VB.NET, Java, C/C++, Delphi,VB
SQLLineage [11]	Where and How	MIT License	2025-02-08	DuckDB, PostgreSQL, MySQL +23 more	none
Tokern [12]	Where and How	MIT License	2021-10-13	PostgreSQL, AWS Redshift and Snowflake	none
Marquez [13]	Where, How and Why	Apache 2.0 License	2024-10-24	none	none
OpenMetadata [14]	Where, How and Why	Apache 2.0 License	2025-04-18	co	none

Table 1. *Fine Grained* lineage tool comparison

3. Marquez: Selected for its active development, adherence to a widely adopted OpenLineage standard (used in tools such as dbt [15]), and its ability to generate lineage across both Python and SQL pipelines.

OpenMetadata was considered as an alternative to Marquez. However, since its lineage ingestion relies on available connectors, this review focused on the native functionality of OpenLineage and Marquez.

4. Underlying SQL Processing Libraries and Usage

Most open-source tools rely on publicly available libraries for automated SQL processing. For example, SQLLineage uses both SQLParse and SQLFluff (see Figure 2).

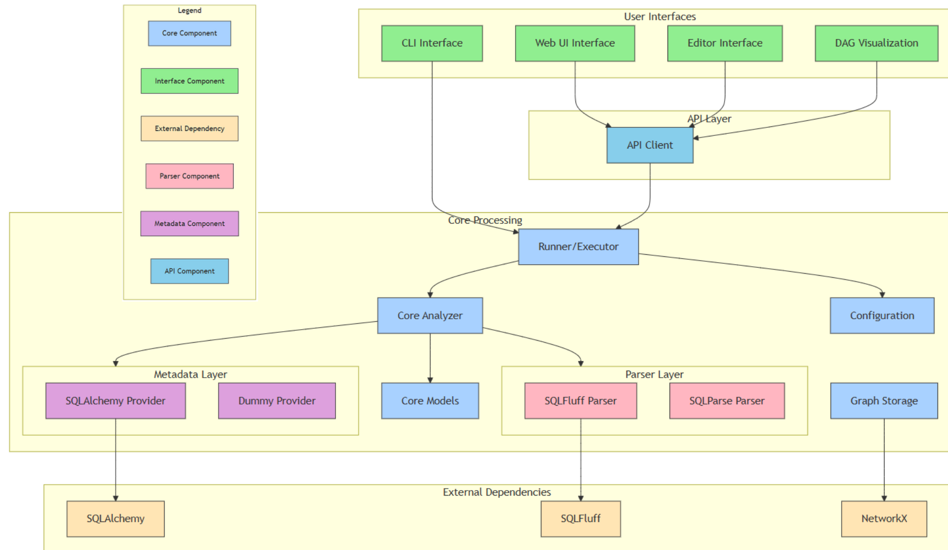


Figure 2. SQLLineage project diagram [16]

These libraries may offer greater capabilities than the data lineage tools that utilize them. By configuring or fully implementing these libraries, one can extend the capabilities of existing tools or even develop a more robust, custom-built solution. Our selection of libraries for review was based on their usage by our chosen tools or recommendations from experienced users.

4.1 SQLFluff

SQLFluff is an open-source, Python-based SQL linter and parser initially developed by Alan Cruickshank during a hackathon at tails.com in late 2019 [17, 18]. SQLFluff's primary goal is quality assurance by ensuring SQL readability and enforcing a consistent coding style. To accomplish this, it incorporates a whitespace-aware parser.

SQLFluff currently supports 26 SQL dialects, including PostgreSQL and DuckDB, and serves as one of the underlying parsers for SQLLineage, generating an *abstract syntax tree* (AST) for it to process (see Figure 3). SQLFluff can be extensively configured to enforce various linting rules and support query templating, beneficial for parsing SQL queries containing `psql` parameters.

```
[L:170, P:   1] | statement:
[L:170, P:   1] |     create_index_statement:
[L:170, P:   1] |         keyword:                  'create'
[L:170, P:   7] |         whitespace:               ' '
[L:170, P:   8] |         keyword:                  'index'
[L:170, P:  13] |         whitespace:               ' '
[L:170, P:  14] |         index_reference:
[L:170, P:  14] |             naked_identifier:
↪   'idx_cda_diagnosis_info_diag_statistical_type'
[L:170, P:  58] |         whitespace:               ' '
[L:170, P:  59] |         keyword:                  'on'
[L:170, P:  61] |                whitespace:               ' '
[L:170, P:  62] |         table_reference:
[L:170, P:  62] |             naked_identifier:     'cda_diagnosis_info'
[L:170, P:  80] |         bracketed:
[L:170, P:  80] |             start_bracket:        '('
[L:170, P:  81] |             [META] indent:
[L:170, P:  81] |             index_element:
[L:170, P:  81] |                 column_reference:
[L:170, P:  81] |                     naked_identifier:
↪   'diag_statistical_type'
[L:170, P:102] |             [META] dedent:
[L:170, P:102] |             end_bracket:          ')'
[L:170, P:103] | statement_terminator:    ';'
[L:170, P:104] | newline:                 '\n'
```

Figure 3. SQLFluff parse example output

4.2 SQLParse

SQLParse is an open-source, Python-based, non-validating SQL parser initially released by Andi Albrecht on April 8, 2009 [19]. It can be used to split, parse, format, and tokenize SQL queries. For instance, the following Python example demonstrates its usage:

```
import sqlparse
sql= """SELECT di.diag_statistical_type FROM tmp_diagnosis_cleaned di"""
print(sqlparse.parse(sql, encoding=None)[0].tokens)
```

This code parses the SQL statement and generates a list of query tokens,

```
[<DML 'SELECT' at 0x7F7927B95300>,
<Whitespace ' ' at 0x7F7927B9B4C0>,
<Identifier 'di.dia...' at 0x7F7927BC28D0>,
<Whitespace ' ' at 0x7F7927B9B640>,
<Keyword 'FROM' at 0x7F7927B9B6A0>,
<Whitespace ' ' at 0x7F7927B9B700>,
<Identifier 'tmp_di...' at 0x7F7927BC2850>]
```

which can then be further analyzed.

Although it explicitly lists eight supported dialects, its general parser and extensible lexer allow support for additional keywords. SQLParse is currently used by SQLLineage, though it is in the process of being deprecated in favor of SQLFluff [20].

4.3 SQLGlot

SQLGlot is a Python-based SQL parser, transpiler, optimizer, and engine created by Toby Mao in 2021 [21]. It supports 24 different dialects, including PostgreSQL and DuckDB. It has no external dependencies and supports 24 different SQL dialects, including PostgreSQL and DuckDB. Its primary use case is formatting and translating SQL queries between different dialects. For example, translating an integer division from DuckDB syntax to PostgreSQL can be done with:

```
sqlglot.transpile("SELECT 1//2", read="duckdb",write="postgres")[0]
```

This produces:

```
['SELECT DIV(1, 2)']
```

which is valid PostgreSQL syntax. Additionally, SQLGlot includes a lineage sub-module capable of generating SQL lineage information from SQL queries. It is widely used by popular frameworks such as dbt[1] and SQLMesh[2].

[1] https://www.getdbt.com/
[2] https://sqlmesh.com/

4.4 OpenLineage Parser

The OpenLineage parser was developed as part of the broader OpenLineage project [22]. Unlike other libraries listed, it is implemented in Rust rather than Python, though it provides Python and Java interfaces. It supports 10 different dialects, including PostgreSQL and a generic SQL dialect, and is used to generate AST for given SQL queries. The parser facilitates the generation of OpenLineage calls, supporting integration into custom data processing workflows. For example, the following Python snippet demonstrates its functionality:

```
sql= """insert into cda_diagnosis_info (
    icd10_diagnosis            ,
    diagnosis_type             )
    select
       md.diag_code,
       di.diag_statistical_type
    from tmp_diagnosis_cleaned di
         left join classifications_schema.diagnoses md
             on upper(di.diag_code) = upper(md.diag_code)"""
meta= parse([sql], dialect="postgres")
print("Input tables are:")
print(meta.in_tables)
print("Output tables are:")
print(meta.out_tables)
print("Column lineage is:")
print(meta.column_lineage)
```

This parses the SQL statement and provides metadata, including input tables, output tables, and column-level lineage information:

```
Input tables are:
[tmp_diagnosis_cleaned, classifications_schema.diagnoses]

Output tables are:
[cda_diagnosis_info]

Column lineage is:
[origin: ("unknown"), name (diag_code) - [origin:
↪ ("classifications_schema.diagnoses"), name (diag_code) ],
↪ origin: ("unknown"), name (diag_statistical_type) - [origin:
↪ ("tmp_diagnosis_cleaned"), name (diag_statistical_type)]]
```

OpenLineage uses this parsing library in integrations, such as with Apache Airflow.

4.5 Comparison of Libraries

All the listed libraries are actively developed but differ in their primary focus. SQLParse and OpenLineage Parser are primarily parsers, SQLFluff emphasizes linting, and SQLGlot specializes in transpiling. A significant differentiator among these libraries is the number and types of supported SQL dialects (see Table 2). Given our project's primary focus on PostgreSQL,

Library	Lineage tool that uses it	Number of supported dialects	PostgreSQL support	DuckDB support
SQLFluff	SQLLineage	16	Yes	Yes
SQLParse	SQLLineage	8	Yes	No
SQLGlot	SQLMesh	24	Yes	Yes
OpenLineage Parser	OpenLineage	10	Yes	No

Table 2. SQL parsing libraries and supported dialects

we concentrated on libraries supporting this dialect during our review. Additionally, DuckDB is an emerging columnar database system known for high-performance data transformation [23]. Due to its syntactic similarity to PostgreSQL, we included DuckDB in our comparison as a potential future technology.

5. Practical Health Data Transformation Example

5.1 Description of the Example

For our practical example, we used a segment of the Tartu University Health Informatics Research Group's pipeline to test data lineage tools with realistic workflows. The example comprises three parts: a PostgreSQL database, a Bash script, a SQL-based transformation pipeline, and a Python-based transformation pipeline using Luigi (see Figure 4).

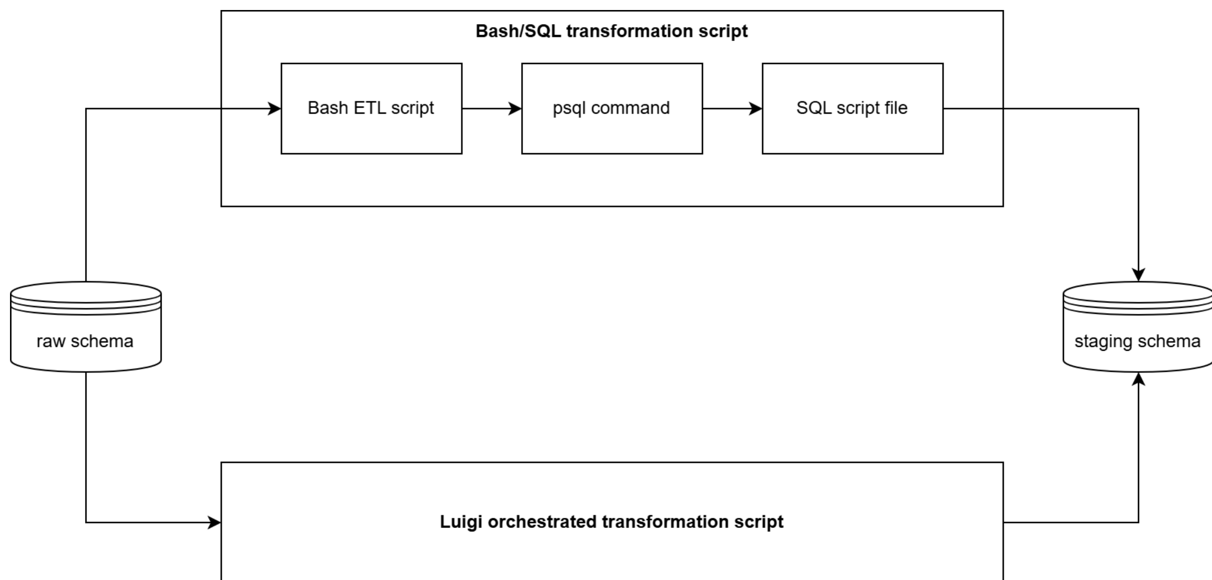


Figure 4. Example data and process flow

5.1.1 Database

The provided example uses a PostgreSQL database to store data for both pipelines. The database is divided into five schemas:

1. `Raw` schema: Holds the original data used as pipeline input;
2. `Stg` schema: Contains intermediate cleaned data (aka `work` schema). In our example, it also holds the final output table;
3. `Mappings` schema: Ensures data consistency by mapping values to standardized equivalents;
4. `Classifications` schema: Stores static classifications, such as the *International Classification of Diseases, version 10* (ICD-10);
5. `Util` schema: Provides utility functions for logging and validation. This schema does contain data and is only used for storing procedures and functions.

Bash scripts are used to run SQL scripts via `psql` to create the demo data and populate these schemas.

5.1.2 Orchestration Pipeline Code

The example code contains two separate workflows:

- First workflow: Uses shell scripts to execute `psql` commands, running SQL scripts to create databases, tables, and initial data.

```
#Defining the script path of the project
SCRIPTPATH="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

#Import helpers to check pre-conditions for import
source "${SCRIPTPATH}/pre-import-checks.sh"

# Setting up the variables used in the code
vars="--set=db_create_role_name==${TARGETDB_ROLE_CREATE}"
vars="$vars --set=work_schema=${TARGETDB_SCHEMA_WORK}"
vars="$vars --set=original_schema=${TARGETDB_SCHEMA_ORIGINAL}"

# Preparing the psql command to run a query file with the listed
↪   variables
exec_sql="psql ${PSQL_HOST_PORT_USER} -d ${TARGETDB} $vars --file"

#Executing the command with the cda_diagnosis_info_load.sql file in
↪   the project
${exec_sql} ${SCRIPTPATH}/code/cda_diagnosis_info_load.sql
```

  These scripts employ variables to inject schema names into templated SQL files (see Figure 5), facilitating configuration across different deployment environments. After generating the source data, further SQL queries transform it. This workflow effectively tests automatic SQL lineage capabilities.

- Second workflow: Employs the Luigi Python package to schedule and execute parallel tasks, performing data transformations required to clean the `diagnosis` table.

```sql
-- A SELECT portion from the cda_diagnosis_info_load.sql file
select distinct id,
    epi_id,
    main_diag_code,
    main_diag_name                as diag_name,
    main_diag_text                as diag_text,
    'main'                        as diag_type,
    main_diag_statistical_type as statistical_type
-- The table schema in here is defined by the
-- ${TARGETDB_SCHEMA_ORIGINAL} variable.
from :original_schema.main_diagnosis
```

Figure 5. SELECT statement excerpt from our example SQL script (*Appendix II*) that uses parameters

5.2 Tokern

5.2.1 Initial Setup Process

Tokern provides a Docker-based deployment method designed to simplify the installation process[3]. Following the official documentation, we granted Tokern access to all required schemas and deployed the Docker containers hosting the web application. Subsequently, we attempted to install the `data-lineage` Python package. We encountered our first challenge here, as the `data-lineage` project has been inactive for several years, leading to installation failures. After troubleshooting, we successfully installed it within a Python 3.9 environment by downgrading the following packages:

- `numpy==1.26.4`;
- `pandas==1.4.4`;
- `pyarrow<8.1`;
- `markupsaf==2.01`;
- `pglast<4`;

[3]https://docs.tokern.io/data-lineage/installation

After the `data-lineage` package was successfully installed, the Tokern Lineage web application became accessible.

5.2.2 Usage of Tools

Upon accessing the Tokern Lineage web application, we observed that the demo data was missing and core functionality was limited. During troubleshooting, we reviewed the deployed Docker containers and noticed that the `tokern_worker` container was stuck in a restart loop. This prompted us to check the container logs, which reported the following error:

```
/docker-entrypoint.sh: 11: exec: rq: not found
```

This is a known issue that has remained unresolved since 2021[4]. Given the number of issues encountered during deployment and initial usage, we decided not to proceed further with this tool. Additionally, considering its abandonment and incompatibility with modern package versions, it poses future security and compatibility risks, further disincentivizing its use.

5.3 OpenLineage and Marquez

5.3.1 Initial Setup Process

OpenLineage and Marquez offer a straightforward, container-based setup that allows rapid initial deployment. To get started, one must clone their repository and execute a single command to launch the container [24]. Although detailed guides for deploying on Amazon Web Services Elastic Kubernetes Service or custom deployments exist, these were outside our test scope. Additionally, the project provides sample data sets, simplifying learning and testing processes.

5.3.2 Usage of Tools

The Marquez project provides an intuitive and visually appealing web interface for reviewing ETL jobs, data flows, and metadata at a granular level. OpenLineage offers a straightforward API to track ETL job metadata, such as affected data points. The level of detail presented by Marquez depends on the specificity provided in the API calls. These API calls can be generated by any system and need to be integrated into the ETL process (see Figure 6). For our review, we used two separate calls to start and complete a transformation job called `test1` (see (*Appendix III* and *Appendix IV*). OpenLineage provides client libraries for Java and Python, simplifying

[4]https://github.com/tokern/data-lineage/issues/89

```
$ curl -X POST http://localhost:5000/api/v1/lineage \
    -i -H 'Content-Type: application/json' \
    -d '{
        "eventType": "START",
        "eventTime": "2020-12-28T19:52:00.001+10:00",
        "run": {
            "runId": "0176a8c2-fe01-7439-87e6-56a1a1b4029f"
        },
        "job": {
            "namespace": "my-namespace",
            "name": "my-job"
        },
        "inputs": [{
            "namespace": "my-namespace",
            "name": "my-input"
        }],
        "producer": "https://github.com/OpenLineage/OpenLineage/
        ↪   blob/v1-0-0/client",
        "schemaURL":
        ↪   "https://openlineage.io/spec/1-0-5/OpenLineage.json#/
        ↪   definitions/RunEvent"
    }'
```

Figure 6. OpenLineage API call example

integration for projects using these languages. However, due to the HTTP-based nature of the API, it remains compatible with most programming environments [25]. After making the necessary API calls, users can utilize Marquez's graphical interface to review the status of transformations and compare them across historical runs.

5.3.3 Extending the Tools

To integrate OpenLineage and Marquez into our pipeline and generate example lineage data, we would need to extend our codebase to include the necessary API calls that trigger lineage generation during each pipeline run. However, we identified no immediate need to modify OpenLineage or Marquez themselves for our use case.

5.3.4 Final Lineage Output

After initial setup and executing API calls to initiate and complete ETL events, the Marquez web interface successfully recorded data activity within the test namespace (see Figure 7). This functionality could be particularly valuable in large, interconnected systems. However, since our example pipeline is based on infrequent batch jobs, it does not fully benefit from this feature.

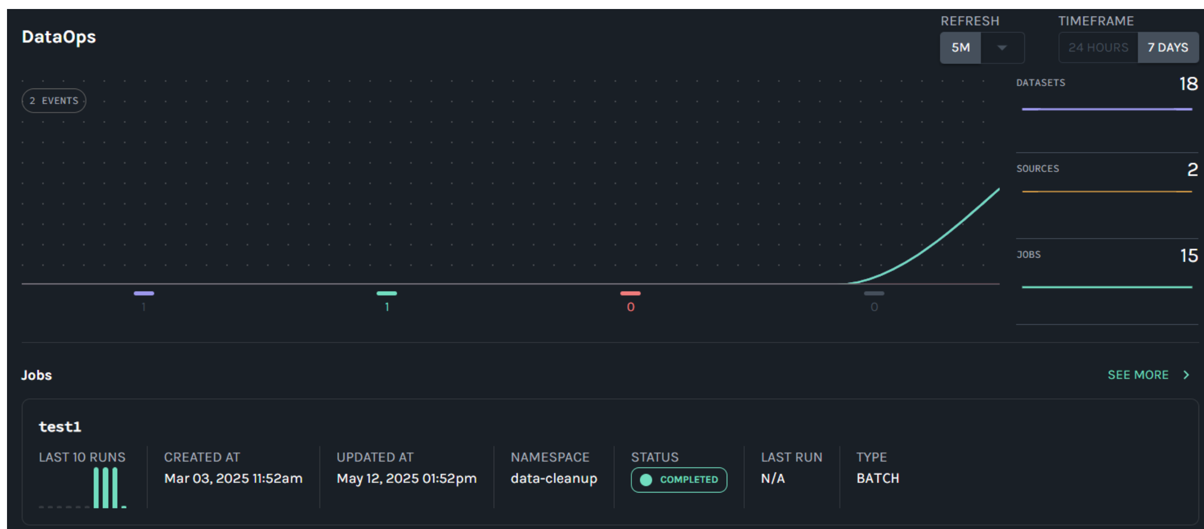


Figure 7. Marquez data operations activity graph

Selecting the test job from the activity list revealed the lineage derivation graph generated by Marquez based on OpenLineage events (see Figure 8). OpenLineage entities adhere to a JSON-based open standard [26], which could be leveraged to extract lineage in a text-based format. This format would support systematic comparison as part of an automated lineage change management system. While this represents a promising direction, implementing such

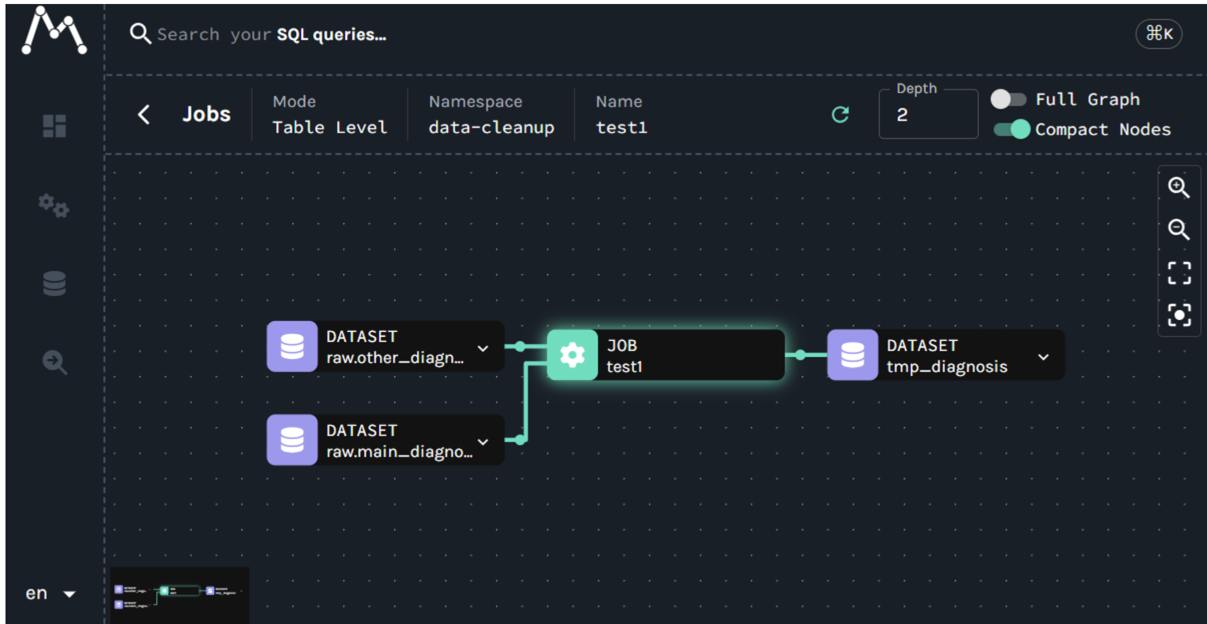


Figure 8. Marquez derivation graph

functionality within our system would require significant development effort. Given the manual nature of our testing, we concluded the process at this stage.

5.4 SQLLineage

5.4.1 Initial Setup Process

To use SQLLineage locally, one installs the Python package, enabling its use as a command-line tool for direct queries or SQL script files. Depending on runtime options, it outputs either text-based or visual lineage information at table or column granularity.

SQLLineage is effective for straightforward SQL-based lineages due to its ease of use, but has limitations, for instance, its inability to process specific SQL statements like CREATE INDEX.

5.4.2 Usage of Tools

After installation, we attempted to generate lineage with the following command:

```
sqllineage -f cda_diagnosis_info_load.sql -l column --dialect=postgres
↪ --silent_mode --sqlalchemy_url=
↪ postgresql+psycopg2://postgres:postgres@127.0.0.1:5433/lineage_exp
```

SQLLineage initially failed to process our SQL scripts due to unsupported `psql` parameters, limiting its immediate usage. This limitation prevents the tool from functioning out of the box with the SQL used in our example. Even after replacing these parameters with fixed values,

errors were encountered due to unsupported statements such as CREATE INDEX and RESET. While the `--silent_mode` flag can be used to skip unsupported statements during command-line lineage generation, these errors persist in the graphical lineage output. Removing these statements enabled successful lineage generation. However, modifying production scripts to suit a tool is impractical, as we would lose flexibility. Given SQLLineage's open-source nature, extending the tool to support our scripts was a viable alternative.

5.4.3 Extending the Tools

SQLLineage utilizes SQLFluff as its primary parsing library. During testing, SQLFluff successfully parsed scripts that did not contain `psql` parameters. Furthermore, SQLFluff's placeholder templater enabled parameter substitution using a `.sqlfluff` configuration file (see Figure 9) [27].

```
[sqlfluff]
templater = placeholder

#parameter format
[sqlfluff:templater:placeholder]
param_style = colon_optional_quotes

#parameter mapping
db_create_role_name = lineageuser
work_schema = stg
original_schema = raw
util_schema = util
classifications_schema = classifications
mappings_schema = mappings
```

Figure 9. Configuration used in the project

After confirming that the parsing library could successfully process the SQL scripts, we reviewed the SQLLineage source code to identify why it failed to handle SQLFluff results for the RESET and CREATE INDEX statements. Within the SQLLineage codebase, multiple extractors are responsible for handling lineage extraction from parsed SQL statements. Each extractor explicitly defines the statement types it supports. If a statement is not listed, SQLLineage does not process it correctly, which can lead to errors during execution.

One specific extractor, `noop.py`, includes statements that do not produce lineage. Since the RESET and CREATE INDEX statements do not affect lineage information, we added these to the supported statement types within the `NoopExtractor` (see Figure 10). After applying

```
from sqlfluff.core.parser import BaseSegment
from sqllineage.core.holders import StatementLineageHolder
from sqllineage.core.parser.sqlfluff.extractors.base import
↪   BaseExtractor
from sqllineage.utils.entities import AnalyzerContext

class NoopExtractor(BaseExtractor):
    """
    Extractor for queries which do not provide any lineage
    """

    SUPPORTED_STMT_TYPES = [
        "delete_statement", "truncate_table", "refresh_statement",
        "cache_table", "uncache_table", "show_statement",
        "describe_statement", "use_statement", "declare_segment",
        "analyze_statement", "add_jar_statement",
        ↪   "create_function_statement",
        "drop_function_statement", "set_statement",
        "create_index_statement", "reset_statement", #Our modification
    ]

    def extract(
        self,
        statement: BaseSegment,
        context: AnalyzerContext,
    ) -> StatementLineageHolder:
        return StatementLineageHolder()
```

Figure 10. Updated SQLLineage code to support CREATE INDEX and RESET statements

these modifications, we rebuilt the SQLLineage Python package and retested it with the original query file. Thanks to these updates to the SQLFluff configuration and SQLLineage source code, the file was successfully processed, resulting in column-level lineage extraction.

5.4.4 Final Lineage Output

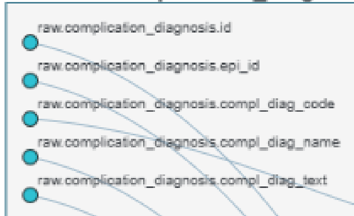
As a result of the setup and extension of SQLFluff and SQLLineage, we were able to automatically generate column-level lineage for the code example. However, this lineage was split into two separate segments. The lineage associated with a nested SELECT statement, named `diag` and used to create the `tmp_diagnosis` table (see Figure 11), appeared separately from the main lineage (see Figure 12). Additionally, the title of the nested SELECT does not display its alias, making it more difficult to correlate the two charts. When switching to table-level lineage, the diagram provides a clearer general overview of the process. However, the nested SELECT remains isolated (see Figure 13).

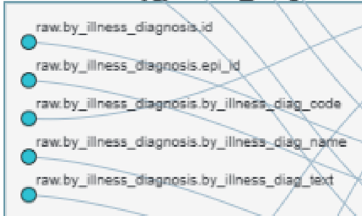
SQLLineage also supports text-based lineage that could be generated from the command line or as part of a Python script. This form of lineage is more limited than the graphical charts (see Figure 15 and Figure 14). For instance, while the text-based table lineage includes intermediate tables, it does not indicate which source tables they are derived from. Nevertheless, this output could be incorporated into an automated, scheduled process to detect changes in process inputs or outputs, thereby further simplifying lineage integration.

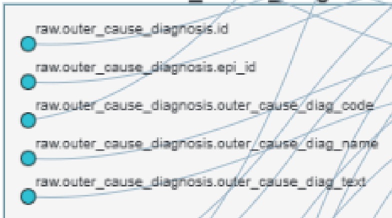
5.5 Summary Deployment Results

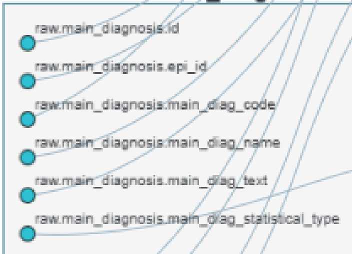
Following configuration and code extension, SQLLineage proved effective for generating column-level lineage from SQL script files. It is well-suited for SQL-based lineage tasks and is relatively straightforward to deploy within an existing pipeline. However, its limitations include a lack of support for non-SQL workflows, reduced expressiveness in text-based output, and missing connections within the derivation graph.

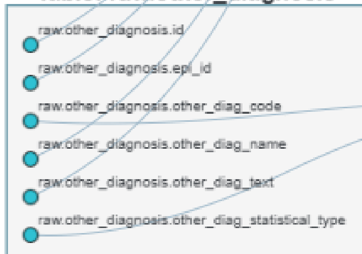
In contrast, OpenLineage and Marquez have a steeper learning curve and lack built-in automation for lineage generation. Nonetheless, it follows an actively maintained open standard, supported by multiple tools. This standard could be used to implement an automated lineage change alerting system. Its robust API facilitates technical integration into modern data pipelines. However, it still requires significant development effort, making it a costly solution for established projects.











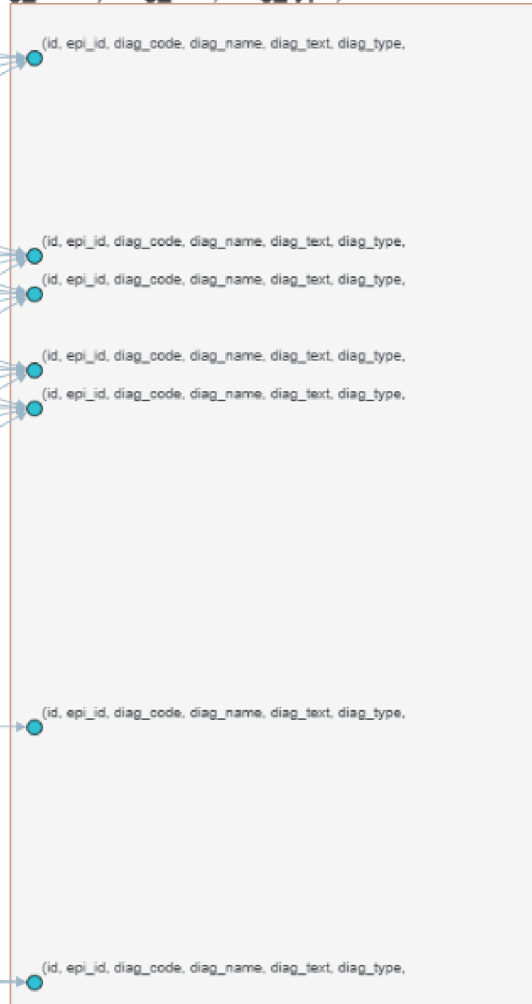


Figure 11. SQLLineage derivation graph of a nested SELECT statement

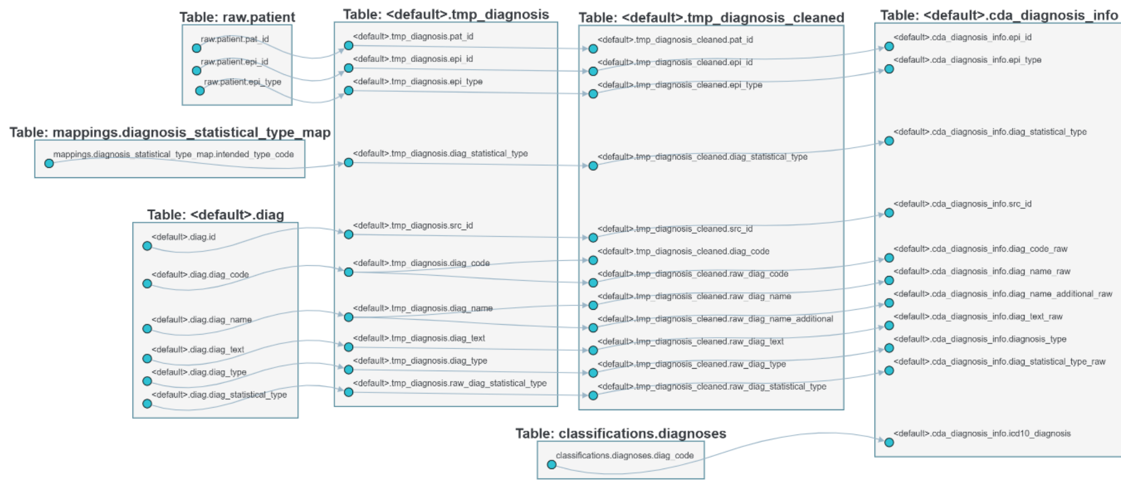


Figure 12. SQLLineage derivation graph of the `cda_diagnosis_info` table

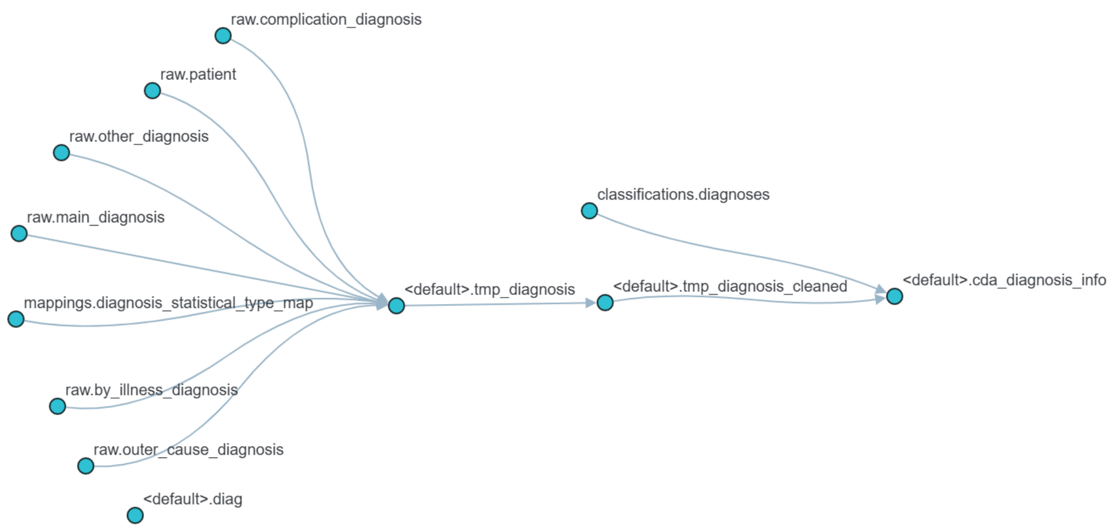


Figure 13. SQLLineage table level derivation graph of the `cda_diagnosis_info` table

```
Statements(#): 17
Source Tables:
    classifications.diagnoses
    mappings.diagnosis_statistical_type_map
    raw.by_illness_diagnosis
    raw.complication_diagnosis
    raw.main_diagnosis
    raw.other_diagnosis
    raw.outer_cause_diagnosis
    raw.patient
Target Tables:
    <default>.cda_diagnosis_info
    <default>.tmp_diagnosis_cleaned
Intermediate Tables:
    <default>.tmp_diagnosis
    <default>.tmp_diagnosis_cleaned
```

Figure 14. SQLLineage tabel level command lineage for the `cda_diagnosis_info` table

```
<default>.cda_diagnosis_info.diag_code_raw <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_code <-
↪  <default>.tmp_diagnosis.diag_code <- <default>.diag.diag_code
<default>.cda_diagnosis_info.diag_name_additional_raw <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_name_additional <-
↪  <default>.tmp_diagnosis.diag_name <- <default>.diag.diag_name
<default>.cda_diagnosis_info.diag_name_raw <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_name <-
↪  <default>.tmp_diagnosis.diag_name <- <default>.diag.diag_name
<default>.cda_diagnosis_info.diag_statistical_type <-
↪  <default>.tmp_diagnosis_cleaned.diag_statistical_type <-
↪  <default>.tmp_diagnosis.diag_statistical_type <-
↪  mappings.diagnosis_statistical_type_map.intended_type_code
<default>.cda_diagnosis_info.diag_statistical_type_raw <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_statistical_type <-
↪  <default>.tmp_diagnosis.raw_diag_statistical_type <-
↪  <default>.diag.diag_statistical_type
<default>.cda_diagnosis_info.diag_text_raw <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_text <-
↪  <default>.tmp_diagnosis.diag_text <- <default>.diag.diag_text
<default>.cda_diagnosis_info.diagnosis_type <-
↪  <default>.tmp_diagnosis_cleaned.raw_diag_type <-
↪  <default>.tmp_diagnosis.diag_type <- <default>.diag.diag_type
<default>.cda_diagnosis_info.epi_id <-
↪  <default>.tmp_diagnosis_cleaned.epi_id <-
↪  <default>.tmp_diagnosis.epi_id <- raw.patient.epi_id
<default>.cda_diagnosis_info.epi_type <-
↪  <default>.tmp_diagnosis_cleaned.epi_type <-
↪  <default>.tmp_diagnosis.epi_type <- raw.patient.epi_type
<default>.cda_diagnosis_info.icd10_diagnosis <-
↪  classifications.diagnoses.diag_code
<default>.cda_diagnosis_info.src_id <-
↪  <default>.tmp_diagnosis_cleaned.src_id <-
↪  <default>.tmp_diagnosis.src_id <- <default>.diag.id
<default>.tmp_diagnosis_cleaned.diag_code <-
↪  <default>.tmp_diagnosis.diag_code <- <default>.diag.diag_code
<default>.tmp_diagnosis_cleaned.pat_id <- <default>.tmp_diagnosis.pat_id
↪  <- raw.patient.pat_id
```

Figure 15. SQLLineage column level command lineage for the `cda_diagnosis_info` table

6. Conclusion

This study reviewed data lineage tools suitable for use in health data transformation pipelines. Our tool review identified Marquez, which uses the OpenLineage standard, and SQLLineage, which is capable of creating automated SQL lineage, as suitable tools for this. We also reviewed the parsing libraries used by these tools and found that the underlying libraries might offer more capabilities than the lineage tools themselves.

During our deployment of OpenLineage and Marquez on the example pipeline, we found it to be a capable tool, straightforward to set up, but costly to integrate into an existing project. It lacks out-of-the-box automation and would require modifications to our existing codebase to include its API calls.

SQLLineage proved easy to use and capable of providing automated SQL lineage, although it did not function immediately out of the box. It required adjustments to the SQLFluff library configuration and an extension of the SQLLineage code. After these developments, it successfully generated a derivation graph from our example SQL code.

Currently, the SQLFluff configuration must be created manually, meaning that any changes to parameters require corresponding manual edits. A potential future would be to automate the generation of this configuration and update it dynamically based on the existing environment. Additionally, SQLLineage could be integrated into an automated deployment system to provide notifications of any data lineage changes. As part of this work, we have created a fork of the SQLLineage repository[5] incorporating the proposed changes, and we intend to follow up with a contribution to the main branch for possible integration.

[5]https://github.com/Veerand/sqllineage

References

[1] Buneman P., Khanna S., and Wang-Chiew T. Why and Where: A Characterization of Data Provenance. *Database Theory — ICDT 2001*. Ed. by Van den Bussche J. and Vianu V. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 316–330.

[2] U.S. Department of Health and Human Services. Summary of the HIPAA Privacy Rule. https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html. (05/08/2025).

[3] European Parliament, Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng. (05/08/2025).

[4] U.S. Department of Health and Human Services. OCR Concludes 2018 with All-Time Record Year for HIPAA Enforcement. https://www.hhs.gov/hipaa/for-professionals/compliance-enforcement/agreements/2018enforcement/index.html. (05/08/2025).

[5] Chessell M. Understanding the Origin of Data. https://poimnotes.blog/2017/03/19/understanding-the-origin-of-data/. (05/14/2025).

[6] Tan W.-c. Provenance in Databases: Past, Current, and Future. *IEEE Data Eng. Bull.* 30 (Jan. 2007), pp. 3–12.

[7] Green T. J., Karvounarakis G., and Tannen V. Provenance semirings. *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '07. Beijing, China: Association for Computing Machinery, 2007, pp. 31–40. DOI: 10.1145/1265530.1265535. https://doi.org/10.1145/1265530.1265535.

[8] Hariharan A., Zhang T., Motz M., and Weinhardt C. Accessible data lineage: A scoping review on open-source data lineage platforms. *ICIS 2024 Proceedings* (2024). https://aisel.aisnet.org/icis2024/data_soc/data_soc/5.

[9] Segner M. Top 5 Open Source Data Lineage Tools (With User Reviews). https://www.montecarlodata.com/blog-open-source-data-lineage-tools/. (05/14/2025).

[10] Gudu Software. General SQL Parser. https://www.sqlparser.com/. (05/08/2025).

[11] Reata and the SQLLineage community. SQLLineage Github repository. https://github.com/reata/sqllineage. (05/08/2025).

[12] Venkatesh R. and Png N. Tokern data-lineage Github repository. https://github.com/tokern/data-lineage. (05/08/2025).

[13] Marquez Project. Marquez - The Complete OpenLineage Solution. https://marquezproject.ai/. (05/08/2025).

[14] OpenMetadata commmunity. OpenMetadata Repository. https://github.com/open-metadata/OpenMetadata. (05/14/2025).

[15] OpenLineage commmunity. OpenLineage dbt Integration documentation. https://openlineage.io/docs/integrations/dbt/. (05/08/2025).

[16] Khaleel A. SQLLineage GitDiagram. https : / / gitdiagram . com / reata / sqllineage. (05/08/2025).

[17] Alan Cruickshank and the SQLFluff commmunity. SQLFluff Github repository. https://github.com/sqlfluff/sqlfluff. (05/08/2025).

[18] dbt Labs, Inc. and Alan Cruickshank. Alan Cruickshank Community Spotlight. https://docs.getdbt.com/community/spotlight/alan-cruickshank. (05/08/2025).

[19] Andi Albrecht and the SQLParse community. SQLParse Github repository. https://github.com/andialbrecht/sqlparse. (05/08/2025).

[20] Reata and the SQLLineage community. SQLLineage Changelog. https://sqllineage.readthedocs.io/en/latest/release_note/changelog.html. (05/08/2025).

[21] Toby Mao and the SQLGlot commmunity. SLGlot Github repository. https://github.com/tobymao/sqlglot. (05/08/2025).

[22] OpenLineage commmunity. OpenLineage Github repository. https://github.com/OpenLineage/OpenLineage/tree/main/integration/sql. (05/08/2025).

[23] DuckDB Foundation. DuckDB - An in-process SQL OLAP Database Management System. https://duckdb.org/. (05/08/2025).

[24] Marquez Project. Marquez Quickstart Guide. https://marquezproject.ai/docs/quickstart/. (05/08/2025).

[25] OpenLineage commmunity. OpenLineage Python Client Library documentation. https://openlineage.io/docs/client/python. (05/08/2025).

[26] OpenLineage commmunity. OpenLineage Dataset Faced Documentation. https://openlineage.io/docs/spec/facets/dataset-facets/. (05/08/2025).

[27] Alan Cruickshank and the SQLFluff commmunity. SQLFluff Placeholder Templater Documentation. https://docs.sqlfluff.com/en/stable/configuration/templating/placeholder.html. (05/08/2025).

Appendices

I. Glossary

Data Lineage - Data lineage is generally defined as a kind of data life cycle that includes the data's origins and where it moves over time. This term can also describe what happens to data as it goes through diverse processes. Data lineage can help with efforts to analyze how information is used and to track key bits of information that serve a particular purpose[6].

Data Provenance - Data provenance is the description of the origins of a piece of data and the process by which it arrived in a database. Data Provenance is sometimes called ”data lineage” [1]

DevOps - The integration and automation of the software development and information technology operations[7]

Pipeline - A set of processing elements connected in series[8]

AI - Artificial Intelligence

AST - Abstract Syntax Tree

API - Application Programming Interface

ETL - Extract, Transform and Load

GDPR - General Data Protection Regulation

HIPAA - Health Insurance Portability and Accountability Act

ICD-10 - International Classification of Diseases, version 10

LLM - Large Language Model

SQL - Structured Query Language

---

[6]https://www.techopedia.com/definition/28040/data-lineage
[7]https://en.wikipedia.org/wiki/DevOps
[8]https://en.wikipedia.org/wiki/Pipeline_(computing)

II. SQL Transformation Script

```
--set working schema
set search_path to :work_schema;
--role for table creating

set role :db_create_role_name;

drop table if exists cda_diagnosis_info cascade;

create table cda_diagnosis_info
(
    id              SERIAL PRIMARY KEY NOT NULL,
    src_id                  INTEGER,
    epi_id                     text,
    epi_type                   text,
    icd10_diagnosis            text,
    diagnosis_type             text,
    diag_statistical_type      text,
    diag_name_additional_raw   text,
    diag_code_raw              text,
    diag_name_raw              text,
    diag_text_raw              text,
    diag_statistical_type_raw text
);

reset role;

--import from original tables
drop table if exists tmp_diagnosis;

create temp table tmp_diagnosis as
select
    diag.id as src_id,
    pat.pat_id,
    pat.epi_id,
    pat.epi_type,
    diag.diag_code,
    diag.diag_name,
    diag.diag_text,
```

```
    diag.diag_type,
    coalesce(map.intended_type_code, '4') as diag_statistical_type,
    diag.diag_statistical_type as raw_diag_statistical_type
from (select distinct id,
    epi_id,
    main_diag_code,
    main_diag_name              as diag_name,
    main_diag_text              as diag_text,
    'main'                      as diag_type,
    main_diag_statistical_type as statistical_type
    from :original_schema.main_diagnosis
    union
    select distinct id,
        epi_id,
        other_diag_code,
        other_diag_name              as diag_name,
        other_diag_text              as diag_text,
        'other'                      as diag_type,
        other_diag_statistical_type as statistical_type
    from :original_schema.other_diagnosis
    union
    select distinct id,
        epi_id,
        outer_cause_diag_code,
        outer_cause_diag_name as diag_name,
        outer_cause_diag_text as diag_text,
        'outer_cause'         as diag_type,
        null                  as statistical_type
    from :original_schema.outer_cause_diagnosis
    union
    select distinct id,
        epi_id,
        by_illness_diag_code,
        by_illness_diag_name as diag_name,
        by_illness_diag_text as diag_text,
        'by_illness'         as diag_type,
        null                 as statistical_type
    from :original_schema.by_illness_diagnosis
    union
    select distinct id,
```

```
        epi_id,
        compl_diag_code,
        compl_diag_name as diag_name,
        compl_diag_text as diag_text,
        'comp'          as diag_type,
        null            as statistical_type
    from :original_schema.complication_diagnosis) as diag (id, epi_id,
    ↪  diag_code, diag_name, diag_text, diag_type,
    ↪  diag_statistical_type)
        left join :original_schema.patient as pat
            on diag.epi_id = pat.epi_id
        left join :mappings_schema.diagnosis_statistical_type_map as map
            on diag.diag_statistical_type = map.reported_type;


--clean the diagnosis code
drop table if exists tmp_diagnosis_cleaned;
create temp table tmp_diagnosis_cleaned as (select di.src_id,
    di.epi_id,
    di.pat_id,
    di.epi_type,
    di.diag_code as raw_diag_code,
    case
       when
           diag_code ~* 'D([A-Z])'
           then
           trim(regexp_replace(diag_code, 'D([A-Z])',
            ↪  substring(diag_code, 2, 1)))
       else
           replace(
                   trim(
                           trim(trailing '+' from
                                trim(trailing '*' from
                                     di.diag_code)
                           )
                   ), '..', '.')
       end      as diag_code,
    case
```

```
    when array_to_string(array_remove(string_to_array(di.diag_name,
     ↪  ';'), (string_to_array(di.diag_name, ';'))[1]), ';') = '' then
     ↪  null
    else trim(array_to_string(array_remove(string_to_array(di.diag_name,
     ↪  ';'), (string_to_array(di.diag_name, ';'))[1]), ';'))
    end as raw_diag_name_additional, --take all info after first
     ↪  semicolon
    di.diag_name as raw_diag_name,
    di.diag_text as raw_diag_text,
    di.diag_type as raw_diag_type,
    di.raw_diag_statistical_type as raw_diag_statistical_type,
    diag_statistical_type
from tmp_diagnosis di);


--fix issue with M21.1 and M21.0
 ↪  (https://pub.etervis.ee/classifications/RHK-10/5)
update tmp_diagnosis_cleaned tdc
set diag_code = 'M21.1'
where raw_diag_code = 'M21.0'
  and lower(raw_diag_name) ~* 'sisse';

update tmp_diagnosis_cleaned tdc
set diag_code = 'M21.0'
where raw_diag_code = 'M21.1'
  and lower(raw_diag_name) ~* 'välja';

--insert to the main table
insert into cda_diagnosis_info (
    src_id                    ,
    epi_id                    ,
    epi_type                  ,
    icd10_diagnosis           ,
    diagnosis_type            ,
    diag_statistical_type     ,
    diag_name_additional_raw  ,
    diag_code_raw             ,
    diag_name_raw             ,
    diag_text_raw             ,
    diag_statistical_type_raw )
```

```
 select
    src_id,
    epi_id,
    epi_type,
    md.diag_code                    as icd10_diagnosis,
    raw_diag_type                   as diagnosis_type,
    di.diag_statistical_type,
    raw_diag_name_additional        as diag_name_additional_raw,
    raw_diag_code                   as diag_code_raw,
    raw_diag_name                   as diag_name_raw,
    raw_diag_text                   as diag_text_raw,
    raw_diag_statistical_type       as diag_statisctical_type_raw
from tmp_diagnosis_cleaned di
    left join :classifications_schema.diagnoses md
        on upper(di.diag_code) = upper(md.diag_code)
;

create index idx_cda_diagnosis_info_epi_id on
↪   cda_diagnosis_info(epi_id);
create index idx_cda_diagnosis_info_epi_type on
↪   cda_diagnosis_info(epi_type);
create index idx_cda_diagnosis_info_icd10_diagnosis on
↪   cda_diagnosis_info(icd10_diagnosis);
create index idx_cda_diagnosis_info_diag_statistical_type on
↪   cda_diagnosis_info(diag_statistical_type);
create index idx_cda_diagnosis_info_diagnosis_type on
↪   cda_diagnosis_info(diagnosis_type);
```

III. OpenLineage Job Start Call

```
curl -L -X POST 'http://localhost:5000/api/v1/lineage' \
-H 'Content-Type: application/json' \
--data-raw '{
        "eventType": "START",
        "eventTime": "2025-05-12T20:52:00.001+10:00",
        "run": {
          "runId": "50000000-0000-7439-87e6-56a1a1b4029f"
        },
```

```
    "job": {
      "namespace": "data-cleanup",
      "name": "test1"
    },
             "context": {
                    "SQL": "select diag.id as src_id from test"
            },
    "inputs": [{
      "namespace": "data-cleanup",
      "name": "raw.main_diagnosis",
              "facets": {
        "schema": {
          "_producer":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/client",
          "_schemaURL":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/spec/OpenLineage.json#/definitions/
          ↪  SchemaDatasetFacet",
          "fields": [
            { "name": "id", "type": "BIGINT"},
            { "name": "epi_id", "type": "VARCHAR"},
            { "name": "main_diag_code", "type": "VARCHAR"},
            { "name": "diag_name", "type": "VARCHAR"},
            { "name": "diag_text", "type": "VARCHAR"},
            { "name": "diag_type", "type": "VARCHAR"},
            { "name": "statistical_type", "type": "VARCHAR"}
          ]
        }
      }
    },
            {
      "namespace": "data-cleanup",
      "name": "raw.other_diagnosis",
              "facets": {
        "schema": {
          "_producer":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/client",
```

```
                "_schemaURL":
                ↪  "https://github.com/OpenLineage/OpenLineage/blob/
                ↪  v1-0-0/spec/OpenLineage.json#/definitions/
                ↪  SchemaDatasetFacet",
                "fields": [
                  { "name": "id", "type": "BIGINT"},
                  { "name": "epi_id", "type": "VARCHAR"},
                  { "name": "other_diag_code", "type": "VARCHAR"},
                  { "name": "other_diag_name", "type": "VARCHAR"},
                  { "name": "diag_text", "type": "VARCHAR"},
                  { "name": "diag_type", "type": "VARCHAR"},
                  { "name": "statistical_type", "type": "VARCHAR"}
                ]
              }
            }
          }],
          "producer": "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/client",
          "schemaURL":
          ↪  "https://openlineage.io/spec/1-0-5/OpenLineage.json#/
          ↪  definitions/RunEvent"
        }'
```

IV. OpenLineage Job Completion Call

```
curl -X POST http://localhost:5000/api/v1/lineage \
  -i -H 'Content-Type: application/json' \
  -d '{
          "eventType": "COMPLETE",
          "eventTime": "2025-05-12T20:52:00.001+10:00",
          "run": {
            "runId": "50000000-0000-7439-87e6-56a1a1b4029f"
          },
          "job": {
            "namespace": "data-cleanup",
            "name": "test1"
          },
          "outputs": [{
            "namespace": "data-cleanup",
```

```
      "name": "tmp_diagnosis",
      "facets": {
        "schema": {
          "_producer":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/client",
          "_schemaURL":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/spec/OpenLineage.json#/definitions/
          ↪  SchemaDatasetFacet",
          "fields": [
            { "name": "src_id", "type": "BIGINT"},
            { "name": "pat_id", "type": "BIGINT"},
            { "name": "epi_id", "type": "BIGINT"},
            { "name": "epi_type", "type": "VARCHAR"},
            { "name": "diag_code", "type": "VARCHAR"},
            { "name": "diag_name", "type": "VARCHAR"},
            { "name": "diag_text", "type": "VARCHAR"},
            { "name": "diag_type", "type": "VARCHAR"},
            { "name": "diag_statistical_type", "type": "VARCHAR"},
            { "name": "raw_diag_statistical_type", "type":
            ↪  "VARCHAR"}
          ]
        },
                    "columnLineage": {
                     "_producer":
                     ↪  "https://github.com/OpenLineage/OpenLineage/
                     ↪  blob/v1-0-0/client",
          "_schemaURL":
          ↪  "https://github.com/OpenLineage/OpenLineage/blob/
          ↪  v1-0-0/spec/OpenLineage.json#/definitions/
          ↪  SchemaDatasetFacet",
                      "fields": {
                            "epi_id": {
                              "inputFields": [
                                    {"namespace": "data-cleanup",
                                    ↪  "name":
                                    ↪  "raw.main_diagnosis",
                                    ↪  "field": "epi_id"},
```

```
                                        {"namespace": "data-cleanup",
                                        ↪   "name":
                                        ↪   "raw.other_diagnosis",
                                        ↪   "field": "epi_id"}
                                  ],
                                  "transformationDescription":
                                  ↪  "identical",
                                  "transformationType": "IDENTITY"
                                }
                          }
                        }
          }
        }
                ],
        "producer": "https://github.com/OpenLineage/OpenLineage/blob/
        ↪  v1-0-0/client",
        "schemaURL":
        ↪  "https://openlineage.io/spec/1-0-5/OpenLineage.json#/
        ↪  definitions/RunEvent"
      }'
```

License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Siim Neljandik**,

*(*author’s name)

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Understanding and Visualizing Data Lineage in Health Data Transformation Pipelines,

   *(*title of thesis)

   supervised by Harry-Anton Talvik;

   *(*supervisor’s name)

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;
3. am aware of the fact that the author retains the rights specified in points 1 and 2;
4. confirm that granting the non-exclusive licence does not infringe other persons’ intellectual property rights or rights arising from the personal data protection legislation.

Siim Neljandik

15/05/2025