sqllineage

Release 1.3.7

Reata

Nov 08, 2022

FIRST STEPS

Never get the hang of a SQL parser? SQLLineage comes to the rescue. Given a SQL command, SQLLineage will tell you its source and target tables, without worrying about Tokens, Keyword, Identified and all the jagons used by a SQL parser.

Behind the scene, SQLLineage uses the fantastic sqlparse library to parse the SQL command, and bring you all the human-readable result with ease.

CHAPTER

ONE

FIRST STEPS

1.1 Getting Started

1.1.1 Install via PyPI

Install the package via `pip` (or add it to your `requirements.txt` file), run:

```
pip install sqllineage
```

1.1.2 Install via GitHub

If you want the latest development version, you can install directly from GitHub:

```
pip install git+https://github.com/reata/sqllineage.git
```

Note: Installation from GitHub (or source code) requires NodeJS/npm for frontend code building, while for PyPI, we already pre-built the frontend code so Python/pip will be enough.

1.1.3 SQLLineage in Command Line

After installation, you will get a *sqllineage* command. It has two major options:

- -e option let you pass a quoted query string as SQL command
- -f option let you pass a file that contains one or more SQL commands

```
$ sqllineage -e "insert into table_foo select * from table_bar union select * from table_
↪baz"
Statements(#): 1
Source Tables:
    <default>.table_bar
    <default>.table_baz
Target Tables:
    <default>.table_foo
```

```
$ sqllineage -f foo.sql
Statements(#): 1
Source Tables:
    <default>.table_bar
    <default>.table_baz
Target Tables:
    <default>.table_foo
```

1.2 Advanced Usage

1.2.1 Multiple SQL Statements

Lineage result combined for multiple SQL statements, with intermediate tables identified

```
$ sqllineage -e "insert into db1.table1 select * from db2.table2; insert into db3.table3␣
↪select * from db1.table1;"
Statements(#): 2
Source Tables:
    db2.table2
Target Tables:
    db3.table3
Intermediate Tables:
    db1.table1
```

1.2.2 Verbose Lineage Result

And if you want to see lineage result for every SQL statement, just toggle verbose option

```
$ sqllineage -v -e "insert into db1.table1 select * from db2.table2; insert into db3.
↪table3 select * from db1.table1;"
Statement #1: insert into db1.table1 select * from db2.table2;
    table read: [Table: db2.table2]
    table write: [Table: db1.table1]
    table cte: []
    table rename: []
    table drop: []
Statement #2: insert into db3.table3 select * from db1.table1;
    table read: [Table: db1.table1]
    table write: [Table: db3.table3]
    table cte: []
    table rename: []
    table drop: []
==========
Summary:
Statements(#): 2
Source Tables:
    db2.table2
Target Tables:
    db3.table3
```

(continues on next page)

(continued from previous page)

```
Intermediate Tables:
    db1.table1
```

1.2.3 Column-Level Lineage

We also support column level lineage in command line interface, set level option to column, all column lineage path will be printed.

```
INSERT OVERWRITE TABLE foo
SELECT a.col1,
       b.col1     AS col2,
       c.col3_sum AS col3,
       col4,
       d.*
FROM bar a
         JOIN baz b
              ON a.id = b.bar_id
         LEFT JOIN (SELECT bar_id, sum(col3) AS col3_sum
                    FROM qux
                    GROUP BY bar_id) c
                   ON a.id = sq.bar_id
         CROSS JOIN quux d;

INSERT OVERWRITE TABLE corge
SELECT a.col1,
       a.col2 + b.col2 AS col2
FROM foo a
         LEFT JOIN grault b
              ON a.col1 = b.col1;
```

Suppose this sql is stored in a file called foo.sql

```
$ sqllineage -f foo.sql -l column
<default>.corge.col1 <- <default>.foo.col1 <- <default>.bar.col1
<default>.corge.col2 <- <default>.foo.col2 <- <default>.baz.col1
<default>.corge.col2 <- <default>.grault.col2
<default>.foo.* <- <default>.quux.*
<default>.foo.col3 <- c.col3_sum <- <default>.qux.col3
<default>.foo.col4 <- col4
```

1.2.4 Lineage Visualization

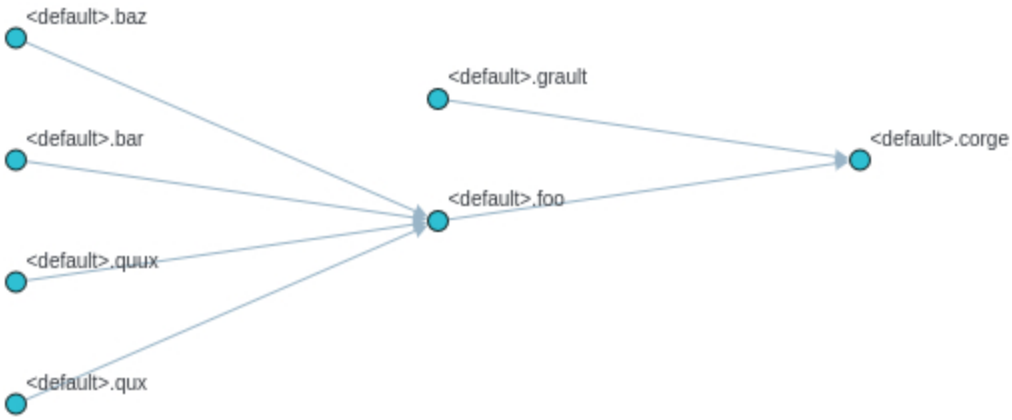
One more cool feature, if you want a graph visualization for the lineage result, toggle graph-visualization option

Still using the above SQL file:

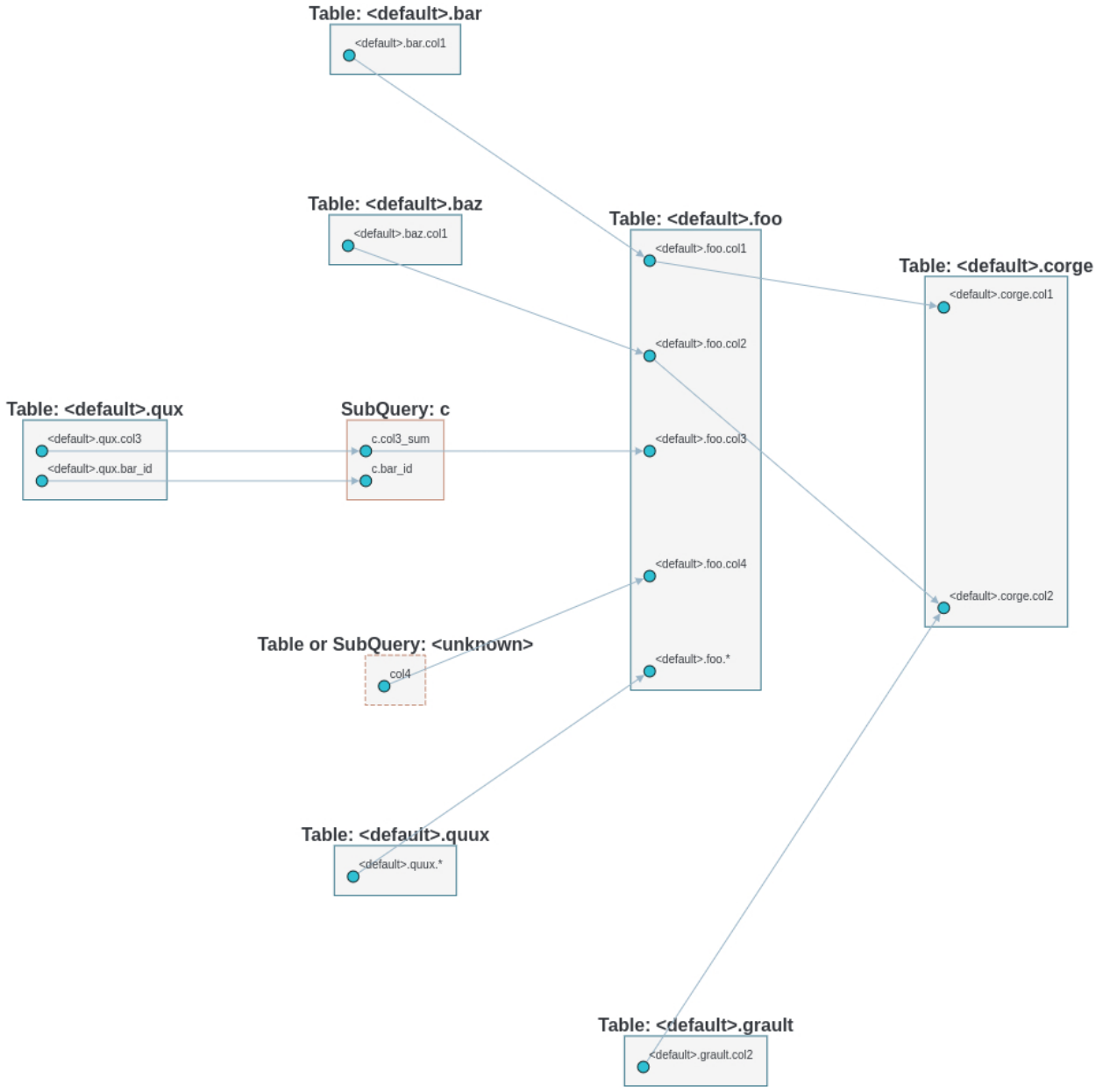
```
sqllineage -g -f foo.sql
```

A webserver will be started, showing DAG representation of the lineage result in browser.

Table-Level Lineage:



Column-Level Lineage:



1.3 Beyond Command Line

Since sqllineage is a Python package, after installation, you can also import it and use the Python API to achieve the same functionality.

```
>>> from sqllineage.runner import LineageRunner
>>> sql = "insert into db1.table11 select * from db2.table21 union select * from db2.
↪table22;"
>>> sql += "insert into db3.table3 select * from db1.table11 join db1.table12;"
>>> result = LineageRunner(sql)
# To show lineage summary
>>> print(result)
Statements(#): 2
Source Tables:
    db1.table12
    db2.table21
    db2.table22
Target Tables:
    db3.table3
Intermediate Tables:
    db1.table11
# To parse all the source tables
>>> for tbl in result.source_tables: print(tbl)
db1.table12
db2.table21
db2.table22
# likewise for target tables
>>> for tbl in result.target_tables: print(tbl)
db3.table13
# To pop up a webserver for visualization
>>> result.draw()
```

CHAPTER

TWO

BEHIND THE SCENE

2.1 Why SQLLineage

2.1.1 How It Starts

Back when I was a data engineer, SQL is something people can't avoid in this industry. I guess it still is since you're reading this. However popular, it doesn't change the fact the SQL code can be nested, verbose and very difficult to read. Oftentimes, I found myself lost in thousands lines of SQL code full of seemingly invaluable business logic. I have to dive deep to understand which tables this SQL script are reading, how they're joined together and the result is writing to which table. That's really painful experience.

Later I became more of a platform engineer, writing all kinds of tools/platforms for data engineer to enhance their efficiency: job orchestration system, metadata management system, to name a few. That's when the SQL Lineage puzzle found me again. I want a table-level, even column-level lineage trace in my metadata management system. Dependency configuration is tedious work in job orchestration. I want my data engineer colleagues to just write the SQL, feed it to my system, and I'll show them the correct jobs they should depend on given their SQL code.

Back then, I wasn't equipped with the right tool though. Without much understanding of parsing, lexical analyser, let alone a real compiler, regular expression was my weapon of choice. And the strategy is simple enough, whenever I see "from" or "join", I extract the word after it as source table. Whenever I see "insert overwrite", a target table must follow upon. I know you must have already figured out several pitfalls that comes with this approach. How about when "from" is in a comment, or even when it is a string value used in where condition? These are all valid points, but somehow, I managed to survive with this approach plus all kinds of if-else logic. Simple as it is, 80% of the time, I got it right.

But to be more accurate, even get it right 100% of the time, I don't think regular expression could do the trick. So I searched the internet, tons of sql parsers, while no handy/friendly tool to use these parsers to analyze the SQL lineage, not at least in Python world. So I thought, hey, why not build one for my own. That's how I decided to start this project. It aims at bridging the gap between a) sql parser which explains the sql in a way that computer (sql engine) could understand, and b) sql developers with the knowledge to write it while without technical know-how for how this sql code is actually parsed, analyzed and executed.

With SQLLineage, it's all just human-readable lineage result.

2.1.2 Broader Use Case

SQL lineage, or data lineage if we include generic non-SQL jobs, is the jewel in the data engineering crown. A well maintained lineage service can greatly ease the pain felt among different roles in a data team.

Here's a few classical use cases for lineage:

- **For Data Producer**
  - **Dependency Recommendation**: recommending dependency for jobs, detecting missing or unnecessary dependency to avoid potential issues.
  - **Impact analysis**: notifying downstream customer when data quality issue happens, tracing back to upstream for root cause analysis, understanding how much impact it would be when changing a table/column.
  - **Development Standard Enforcement**: detecting anti-pattern like one job producing multiple production tables, or temporary table/view created without being accessed later.
  - **Job ETA Prediction and Alert**: predicting table delay and potential SLA miss with job running information.
- **For Data Governor**
  - **Table/Column Lifecycle**: identifying unused tables/columns, and retiring it.
  - **GDPR Compliance**: propagating the PII columns tag along the lineage path, easing the manually tagging process. Foundation for later PII encryption and GDPR deletion.
- **For Data Consumer**
  - **Understanding data flow**: discovering table, understanding the table description as well as the table flow.
  - **Verify business logic**: making sure the information I use is sourcing from the correct dataset with correct transformation.

2.2 How Does SQLLineage Work

Basically a sql parser will parse the SQL statement(s) into AST (Abstract Syntax Tree), which according to wikipedia, is a tree representation of the abstract syntactic structure of source code (in our case, SQL code, of course). This is where SQLLineage takes over.

With AST generated, SQLLineage will traverse through this tree, apply some pre-defined rules, so as to extract the part we're interested in. With that being said, SQLLineage is an AST application, while there's actually more you can do with AST:

- **born duty of AST: the starting point for optimization.** In compiler world, machine code, or optionally IR (Intermediate Representation), will be generated based on the AST, and then code optimization, resulting in an optimized machine code. In data world, it's basically the same thing with different words, and different optimization target. AST will be converted to query execution plan for query execution optimization, using strategy like RBO(Rule Based Optimization) or CBO(Cost Based Optimization), so that database/data warehouse query engine can have an optimized physical plan for execution.
- **linter**: quoting wikipedia, linter is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs. Oftentimes it's used interchangeably with a code formatter. Famous tools like flake8 for Python, ESLint for JavaScript. Golang even provide an official gofmt program in their standard library. Meanwhile, although not yet widely adopted in data world, we can also lint SQL code. sqlfluff is such an great tool. Guess how it works to detect a smelly "*SELECT* *" or a mixture of leading and trailing commas. The answer is AST!

- **transpiler**: This use case is most famous in JavaScript world, where they're proactively using syntax defined in latest language specification which is not supported by mainstream web browsers yet. Quote from its offical document, Babel is capable of "converting ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers". Babel exists in frontend because there's less control over runtime choice compared to backend developers, you need both browser to support it and user to upgrade their browser. In data world, we also see similar requirements for SQL syntax difference between different SQL dialect. sqlglot is such an project, which can help you "translate" for example SQL written in Hive to Presto. All there are not possible without AST.
- **structure analysis**: IDE leverages this a lot. Scenarios like duplicate code detection, code refactor. Basically this is to analyze the code structure. SQLLineage also falls into this category.

sqlparse is the underlying parser SQLLineage uses to get the AST. It gives a simple example to extract table names, through which you can get a rough idea of how SQLLineage works. At the core is when a token is Keyword and its value is "FROM", then the next token will either be subquery or table. For subquery, we just recursively calling extract function. For table, there's a way to get its name.

Warning: This is just an over-simplified explanation. In reality, we could easily see `Comment` coming after "FROM", or subquery without alias (valid syntax in certain SQL dialect) mistakenly parsed as `Parenthesis`. These are all corner cases we should resolve in real world.

Note: Strictly speaking, sqlparse is generating a parse tree instead of an abstract syntax tree. There two terms are often used interchangeably, and indeed they're similar conceptually. They're both tree structure in slightly different abstraction layer. In the AST, information like comments and grouping symbols (parenthesis) are not represented. Removing comment doesn't change the code logic and parenthesis are already implicitly represented by the tree structure.

Some other simple rules in SQLLineage:

1. Things go after Keyword **"FROM"**, all kinds of **"JOIN"** will be source table.
2. Things go after Keyword **"INTO"**, **"OVERWRITE"**, **"TABLE"**, **"VIEW"** will be target table. (Though there are exceptions like drop table statement)
3. Things go after Keyword **"With"** will be CTE (Common Table Expression).
4. Things go after Keyword **"SELECT"** will be column(s).

The rest thing is just tedious work. We collect all kinds of sql, handle various edge cases and make these simple rules robust enough.

That's it for single statement SQL lineage analysis. For multiple statements SQL, it requires some more extra work to assemble the lineage from single statements.

We choose a DAG based data structure to represent multiple statements SQL lineage. Table/View will be vertex in this graph while a edge means data in source vertex table will contribute to data in target vertex table. In column-level lineage, the vertex is a column. Every single statement lineage result will contain table/column read and table/column write information, which will later be combined into this graph. With this DAG based data structure, it is also very easy to visualize lineage.

2.3 DOs and DONTs

SQLLineage is a static SQL code lineage analysis tool, which means we will not try to execute the SQL code against any kinds of server. Instead, we will just look at the code as text, parse it and give back the result. No client/server interaction.

2.3.1 DOs

- SQLLineage will continue to support most commonly used SQL system. Make best effort to be compatible.
- SQLLineage will stay mainly as a command line tool, as well as a Python utils library.

2.3.2 DONTs

- Column-level lineage will not be 100% accurate because that would require metadata information. However, there's no unified metadata service for all kinds of SQL systems. For the moment, in column-level lineage, column-to-table resolution is conducted in a best-effort way, meaning we only provide possible table candidates for situation like `select *` or `select col from tab1 join tab2`.
- Likewise for Partition-level lineage. Until we find a way to not involve metadata service, we will not go for this.

Note: 100% accurate Column-level lineage is still do-able if we can provide some kind of a plugin system for user to register their metadata instead of us maintaining it. Let's see what will happen in future versions.

2.3.3 Static Code Analysis Approach Explained

This is the typical flow of how SQL is parsed and executed from compiling perspective:

1. Lexical Analysis: transform SQL string into token stream.
2. Parsing: transform token stream into unresolved AST.
3. Semantic Analysis: annotate unresolved AST with real table/column reference using database catalog.
4. Optimize with Intermediate Representation: transform AST to execution plan, and optimize with predicate push-down, column pruning, boolean simplification, limit combination, join strategy, etc.
5. Code Gen: This only makes sense in distributed SQL system. Generating primitive on underlying computing framework, like MapReduce Job, or RDD operation, based on the optimized execution plan.

Note: Semantic analysis is a compiler term. In data world, it's often referred to as catalog resolution. For some systems, unresolved AST is transformed to unsolved execution plan first. With catalog resolution, a resolved execution plan is then ready for later optimization phases.

SQLLineage is working on the abstraction layer of unresolved AST, right after parsing phase is done. The good side is that SQLLineage is dialect-agnostic and able to function without database catalog. The bad side is of course the inaccuracy on column-level lineage when we don't know what's behind `select *`.

The alternative way is starting the lineage analysis on the abstraction layer of resolved AST, or execution plan. That ties lineage analysis tightly with the SQL system so it won't function without a live connection to database. But that will give user an accurate result and the source code of database can be used to save a lot of coding effort.

To combine the good side of both approaches, in the long term, SQLLineage will introduce an optional resolution phase, followed by the current unresolved lineage result, where user can register metadata information in a programmatic way.

2.4 Column-Level Lineage Design

2.4.1 Key Design Principles

- **sqllineage will stay primarily as a static code analysis tool**, so we must tolerate information missing when doing column-level lineage. Maybe somewhere in the future, we can provide some kind of plugin mechanism to register metadata as a supplement to refine the lineage result, but in no way will we depend solely on metadata.
- **Don't create column-level lineage DAG to be a separate graph from table-level DAG**. There should be one unified DAG. Either 1) we build a DAG to the granularity of column so with some kind of transformation, we can derive table-level DAG from it. To use an analogy of relational data, it's like building a detail table, with the ability to aggregate up to a summary table. Or 2) we build a property graph (see JanusGraph docs for reference), with Table and Column as two kinds of Vertex, and two type of edges, one for column to table relationship mapping, one for column-level as well as table-level lineage.

2.4.2 Questions Before Implementation

1. **What's the data structure to represent Column-Level Lineage?** Currently we're using `DiGraph` in library `networkx` to represent Table-Level Lineage, with table as vertex and table-level lineage as edge, which is pretty straight forward. After changing to Column-Level, what's the plan?

Answer: See design principle for two possible data structure. I would prefer property graph for this.

2. **How do we deal with `select *` ?** In following case, we don't know which columns are in tab2

```
INSERT OVERWRITE tab1
SELECT * FROM tab2;
```

Answer: Add an virtual column * for each Table as a special column. So the lineage is tab2.* -> tab1.*

3. **How do we deal with column without table/alias prefix in case of join?** In following case, we don't know whether col2 is coming from tab2 or tab3

```
INSERT OVERWRITE tab1
SELECT col2
FROM tab2
JOIN tab3
ON tab2.col1 = tab3.col1
```

Answer: dd two edges, tab2.col2 -> tab1.col2, tab3.col2 -> tab1.col2. Meanwhile, these two edges should be marked so that later in visualization, they can be drawn differently, like in dot line.

2.4.3 Implementation Plan

With 6308b50 splitting the logic into different handlers, we now have SourceHandler, TargetHandler and CTEHandler to handle table level lineage. They're subclass of NextTokenBaseHandler, an abstract class to address an extract pattern when a specified token indicates we should extract something from next token.

A newly introduced ColumnHandler will also be based on NextTokenBaseHandler (column token followed by keyword SELECT) plus a end-of-(sub)query hook. Because only until end of query could we know all the source tables. If we don't have all the source tables and their alias, we can't assign the column to table correctly.

> **Warning:** To handle UNION clause, ColumnHandler is now merged into SourceHandler, due to the fact that we need source tables info breaking down into sub-statement level, end of the whole query would be to late.

Steps for Full Implementation

1. Atomic column logic handling: alias, case when, function, expression, etc.
2. Subquery recognition and lineage transition from subquery to statement
3. Column to table assignment in case of table join
4. Assemble Statement Level lineage into multiple statements DAG.

Note: Column-Level lineage is now released with v1.3.0

Why SQLLineage
: The motivation of writing SQLLineage

How Does SQLLineage Work
: The inner mechanism of SQLLineage

DOs and DONTs
: Design principles for SQLLineage

CHAPTER

THREE

BASIC CONCEPTS

3.1 LineageRunner

LineageRunner is the entry point for SQLLineage core processing logic. After parsing command-line options, a string representation of SQL statements will be fed to LineageRunner for processing. From a bird's-eye view, it contains three steps:

1. Calling `sqlparse.parse` function to parse string-base SQL statements into a list of `sqlparse.sql.Statement`
2. Calling *sqllineage.core.analyzer.LineageAnalyzer* to analyze each `sqlparse.sql.Statement` and return a list of *sqllineage.core.holders.StatementLineageHolder* .
3. Calling *sqllineage.core.holders.SQLLineageHolder.of* function to assemble the list of *sqllineage.core.holders.StatementLineageHolder* into one *sqllineage.core.holders.SQLLineageHolder*.

sqllineage.core.holders.SQLLineageHolder then will serve for lineage summary, in text or in visualization form.

3.1.1 sqllineage.runner.LineageRunner

class sqllineage.runner.**LineageRunner**(*sql: str*, *encoding: Optional[str] = None*, *verbose: bool = False*, *draw_options: Optional[Dict[str, str]] = None*)

The entry point of SQLLineage after command line options are parsed.

Parameters

- **sql** – a string representation of SQL statements.
- **encoding** – the encoding for sql string
- **verbose** – verbose flag indicate whether statement-wise lineage result will be shown

__str__(***kwargs*)

Return str(self).

draw() → None

to draw the lineage directed graph

print_column_lineage() → None

print column level lineage to stdout

print_table_lineage() → None

print table level lineage to stdout

3.1.2 sqllineage.cli.main

sqllineage.cli.**main**(*args=None*) → None

The command line interface entry point.

Parameters

args – the command line arguments for sqllineage command

3.2 LineageAnalyzer

LineageAnalyzer contains the core processing logic for one-statement SQL analysis.

At the core of analyzer is all kinds of `sqllineage.core.handlers` to handle the interested tokens and store the result in `sqllineage.core.holders`.

3.2.1 LineageAnalyzer

class sqllineage.core.analyzer.**LineageAnalyzer**

SQL Statement Level Lineage Analyzer.

analyze(*stmt: Statement*) → *StatementLineageHolder*

to analyze the Statement and store the result into `sqllineage.holders.StatementLineageHolder`.

Parameters

stmt – a SQL statement parsed by *sqlparse*

3.2.2 SourceHandler

class sqllineage.core.handlers.source.**SourceHandler**

Source Table & Column Handler.

3.2.3 TargetHandler

class sqllineage.core.handlers.target.**TargetHandler**

Target Table Handler.

3.2.4 CTEHandler

class sqllineage.core.handlers.cte.**CTEHandler**

Common Table Expression (With Queries) Handler.

3.3 LineageHolder

LineageHolder is an abstraction to hold the lineage result analyzed by LineageAnalyzer at different level.

At the bottom, we have `sqllineage.core.holder.SubQueryLineageHolder` to hold lineage at subquery level. This is used internally for *sqllineage.core.analyzer.LineageAnalyzer*, which generate `sqllineage.core.holder.StatementLineageHolder` as the result of lineage at SQL statement level. And to assemble multiple `sqllineage.core.holder.StatementLineageHolder` into a DAG based data structure serving for the final output, we have *sqllineage.core.holders.SQLLineageHolder*

3.3.1 SubQueryLineageHolder

class sqllineage.core.holders.**SubQueryLineageHolder**

SubQuery/Query Level Lineage Result.

SubQueryLineageHolder will hold attributes like read, write, cte

Each of them is a Set[`sqllineage.models.Table`].

This is the most atomic representation of lineage result.

3.3.2 StatementLineageHolder

class sqllineage.core.holders.**StatementLineageHolder**

Statement Level Lineage Result.

Based on SubQueryLineageHolder, StatementLineageHolder holds extra attributes like drop and rename

For drop, it is a Set[`sqllineage.models.Table`].

For rename, it a Set[Tuple[`sqllineage.models.Table`, `sqllineage.models.Table`]], with the first table being original table before renaming and the latter after renaming.

3.3.3 SQLLineageHolder

class sqllineage.core.holders.**SQLLineageHolder**(*graph: DiGraph*)

The combined lineage result in representation of Directed Acyclic Graph.

Parameters
graph – the Directed Acyclic Graph holding all the combined lineage result.

property table_lineage_graph: DiGraph

The table level DiGraph held by SQLLineageHolder

property column_lineage_graph: DiGraph

The column level DiGraph held by SQLLineageHolder

property source_tables: Set[*Table*]

a list of source `sqllineage.models.Table`

property target_tables: Set[*Table*]

a list of target `sqllineage.models.Table`

property intermediate_tables: Set[*Table*]

a list of intermediate `sqllineage.models.Table`

static of(**args:* StatementLineageHolder)

To assemble multiple `sqllineage.holders.StatementLineageHolder` into `sqllineage.holders.SQLLineageHolder`

3.4 LineageModels

Several data classes in this module.

3.4.1 Schema

class sqllineage.core.models.**Schema**(*name: str = '<default>'*)

Data Class for Schema

Parameters

name – schema name

3.4.2 Table

class sqllineage.core.models.**Table**(*name: str*, *schema: ~sqllineage.core.models.Schema = Schema: <default>*, ***kwargs*)

Data Class for Table

Parameters

- **name** – table name
- **schema** – schema as defined by *Schema*

3.4.3 SubQuery

class sqllineage.core.models.**SubQuery**(*token: Parenthesis*, *alias: Optional[str]*)

Data Class for SubQuery

Parameters

- **token** – subquery token
- **alias** – subquery name

3.4.4 Column

class sqllineage.core.models.**Column**(*name: str*, ***kwargs*)

Data Class for Column

Parameters

- **name** – column name
- **parent** – *Table* or *SubQuery*
- **kwargs** –

LineageRunner

LineageRunner: The entry point for SQLLineage

LineageAnalyzer

LineageAnalyzer: The core functionality of analyze one SQL statement

LineageHolder

LineageCombiner: To hold lineage result at different level

LineageModels

The data classes for SQLLineage

CHAPTER

FOUR

RELEASE NOTE

4.1 Changelog

4.1.1 v1.3.7

Date
Oct 22, 2022

Enhancement

- migrate demo site off Heroku to GitHub Pages (#288)
- remove flask-related dependencies by implementing a wsgi app (#287)

Bugfix

- exception with VALUES clause (#292)
- exception with Presto unnest function (#272)
- exception with snowflake generator statement (#214)

4.1.2 v1.3.6

Date
Aug 28, 2022

Enhancement

- support MySQL RENAME TABLE statement (#267)
- auto deploy to Heroku with GitHub Actions (#232)

Bugfix

- handling parenthesis around subquery between union (#270)
- unable to extract alias of columns using function with CTAS (#253)
- exception when using lateral view (#225)

4.1.3 v1.3.5

Date
May 10, 2022

Enhancement

- support parsing column in cast/try_cast with function (#254)
- support parsing WITH for bucketing in Trino (#251)

Bugfix

- incorrect column lineage with nested cast (#240)
- column lineages from boolean expression (#236)
- using JOIN with ON/USING keyword fails to determine source tables when followed by a parenthesis (#233)
- failure to handle multiple lineage path for same column (#228)

4.1.4 v1.3.4

Date
March 6, 2022

Enhancement

- update black to stable version (#222)

Bugfix

- table/column lineage mixed up for self dependent SQL (#219)
- problem with SELECT CAST(CASE WHEN . . . END AS DECIMAL(M,N)) AS col_name (#215)
- failed to parse source table from subquery with more than one parenthesis (#213)

4.1.5 v1.3.3

Date
December 26, 2021

Enhancement

- smarter column-to-table resolution using query context (#203)

Bugfix

- column lineage for union operation (#207)
- subquery in where clause not parsed for table lineage (#204)

4.1.6 v1.3.2

Date
December 12, 2021

Enhancement

- support optional AS keyword in CTE (#198)
- support referring to a CTE in subsequent CTEs (#196)
- support for Redshift 'copy from' syntax (#164)

4.1.7 v1.3.1

Date
December 5, 2021

Enhancement

- test against Python 3.10 (#186)

Bugfix

- alias parsed as table name for column lineage using ANSI-89 Join (#190)
- CTE parsed as source table when referencing column from cte using alias (#189)
- window function with parameter parsed as two columns (#184)

4.1.8 v1.3.0

Date
November 13, 2021

Feature

- Column-Level Lineage (#103)

Bugfix

- SHOW CREATE TABLE parsed as target table (#167)

4.1.9 v1.2.4

Date
June 14, 2021

Enhancement

- highlight selected node and its ancestors as well as children recursively (#156)
- add support for database.schema.table as identifier name (#153)
- add support for swap_partitions_between_tables (#152)

4.1.10 v1.2.3

Date
May 15, 2021

Enhancement

- lineage API response exception handling (#148)

4.1.11 v1.2.2

Date
May 5, 2021

Bugfix

- resize dragger remain on the UI when drawer is closed (#145)

4.1.12 v1.2.1

Date
May 3, 2021

Enhancement

- option to specify hostname (#142)
- re-sizable directory tree drawer (#140)
- async loading for directory tree in frontend UI (#138)

4.1.13 v1.2.0

Date
April 18, 2021

Feature

- A Full Fledged Frontend Visualization App (#118)
- Use TPC-DS Queries as Visualization Example (#116)

Enhancement

- Unit Test Failure With sqlparse==0.3.0, update dependency to be >=0.3.1 (#117)
- contributing guide (#14)

4.1.14 v1.1.4

Date
March 9, 2021

Bugfix

- trim function with from in arguments (#127)

4.1.15 v1.1.3

Date
February 1, 2021

Bugfix

- UNCACHE TABLE statement parsed with target table (#123)

4.1.16 v1.1.2

Date
January 26, 2021

Bugfix

- Bring back draw method of LineageRunner to avoid backward incompatible change (#120)

4.1.17 v1.1.1

Date
January 24, 2021

Bugfix

- SQLLineageException for Multiple CTE Subclauses (#115)

4.1.18 v1.1.0

Date
January 17, 2021

Feature

- A new JavaScript-based approach for visualization, drop dependency for graphviz (#94)

Enhancement

- Test against Mac OS and Windows (#87)

Bugfix

- buckets parsed as table name for Spark bucket table DDL (#111)
- incorrect result for update statement (#105)

4.1.19 v1.0.2

Date
November 17, 2020

Enhancement

- black check in CI (#99)
- switch to GitHub Actions for CI (#95)
- test against Python 3.9 (#84)

Bugfix

- cartesian product exception with ANSI-89 syntax (#89)

4.1.20 v1.0.1

Date
October 17, 2020

Enhancement

- remove upper bound for dependencies (#85)

4.1.21 v1.0.0

Date
September 27, 2020

New Features

- a detailed documentation hosted by readthedocs (#81)

Enhancement

- drop support for Python 3.5 (#79)

4.1.22 v0.4.0

Date
August 29, 2020

New Features

- DAG based lineage representation with visualization functionality (#55)

Enhancement

- replace print to stderr with logging (#75)
- sort by table name in LineageResult (#70)
- change schema default value from <unknown> to <default> (#69)
- set up Github actions for PyPi publish (#68)

4.1.23 v0.3.0

Date
July 19, 2020

New Features

- statement granularity lineage result (#32)
- schema aware parsing (#20)

Enhancement

- allow user to specify combiner (#64)
- trim leading comment for statement in verbose output (#57)
- add mypy as static type checker (#50)
- add bandit as security issue checker (#48)
- enforce black as code formatter (#46)
- dedicated Table/Partition/Column Class (#31)
- friendly exception handling (#30)

Bugfix

- subquery without alias raises exception (#62)
- refresh table and cache table should not count as target table (#59)
- let user choose whether to filter temp table or not (#23)

4.1.24 v0.2.0

Date
April 11, 2020

Enhancement

- test against Python 3.8 (#39)

Bugfix

- comment in line raise AssertionError (#37)
- white space in left join (#36)
- temp table checking (#35)
- enable case-sensitive parsing (#34)
- support for create table like statement (#29)
- special treatment for DDL (#28)
- empty statement return (#25)
- drop table parsed as target table (#21)
- multi-line sql causes AssertionError (#18)
- subquery mistake alias as table name (#16)

4.1.25 v0.1.0

Date
July 26, 2019

New Features

- stable command line interface (#2)

Enhancement

- combine setup.py and requirements.txt (#6)
- combine tox and Travis CI (#5)
- table-wise lineage with sufficient test cases (#4)
- a startup docs for sqllineage's usage (#3)
- pypi badges in README (#1)

4.1.26 v0.0.1

Date
June 16, 2019

New Features

initial public release

Changelog
See what's new for each SQLLineage version

INDEX

Symbols

A

C

D

I

L

M

O

P

S

T