sqllineage

Release 1.5.2

Reata

Apr 07, 2024

FIRST STEPS

1 First steps **3**
- 1.1 Getting Started . . . 3
- 1.2 Advanced Usage . . . 4
- 1.3 Beyond Command Line . . . 8

2 Gear Up **11**
- 2.1 Configuration . . . 11
- 2.2 MetaData . . . 13

3 Behind the scene **15**
- 3.1 Why SQLLineage . . . 15
- 3.2 How Does SQLLineage Work . . . 16
- 3.3 DOs and DONTs . . . 17
- 3.4 Column-Level Lineage Design . . . 18
- 3.5 Dialect-Awareness Lineage Design . . . 19

4 Basic concepts **23**
- 4.1 Runner . . . 23
- 4.2 Analyzer . . . 24
- 4.3 Holder . . . 25
- 4.4 Model . . . 26
- 4.5 MetaDataProvider . . . 27

5 Release note **29**
- 5.1 Changelog . . . 29

Index **45**

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 pluggable leverages parser library sqlfluff and sqlparse to parse the SQL command, analyze the AST, stores the lineage information in a graph (using graph library networkx), 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 statements
- -f option let you pass a file that contains SQL statements

```
$ 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 is combined from 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 for each 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 Dialect-Awareness Lineage

By default, sqllineage use *ansi* dialect to parse and validate your SQL. However, some SQL syntax you take for granted in daily life might not be in ANSI standard. In addition, different SQL dialects have different set of SQL keywords, further weakening sqllineage's capabilities when keyword used as table name or column name. To get the most out of sqllineage, we strongly encourage you to pass the dialect to assist the lineage analyzing.

Take below example, *INSERT OVERWRITE* statement is only supported by big data solutions like Hive/SparkSQL, and *MAP* is a reserved keyword in Hive thus can not be used as table name while it is not for SparkSQL. Both ansi and hive dialect tell you this causes syntax error and sparksql gives the correct result:

```
$ sqllineage -e "insert overwrite table map select * from foo"
...
sqllineage.exceptions.InvalidSyntaxException: This SQL statement is unparsable, please␣
↪check potential syntax error for SQL

$ sqllineage -e "insert overwrite table map select * from foo" --dialect=hive
...
sqllineage.exceptions.InvalidSyntaxException: This SQL statement is unparsable, please␣
↪check potential syntax error for SQL

$ sqllineage -e "insert overwrite table map select * from foo" --dialect=sparksql
Statements(#): 1
Source Tables:
    <default>.foo
Target Tables:
    <default>.map
```

Use *sqllineage --dialects* to see all available dialects.

1.2.4 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 INTO 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
```

(continues on next page)

(continued from previous page)

```
        CROSS JOIN quux d;

INSERT INTO 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 test.sql

```
$ sqllineage -f test.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
```

1.2.5 MetaData-Awareness Lineage

By observing the column lineage generated from previous step, you'll possibly notice that:

1. *<default>.foo.** <- *<default>.quux.**: the wildcard is not expanded.
2. *<default>.foo.col4* <- *col4*: col4 is not assigned with source table.

It's not perfect because we don't know the columns encoded in * of table *quux*. Likewise, given the context, col4 could be coming from *bar*, *baz* or *quux*. Without metadata, this is the best sqllineage can do.

User can optionally provide the metadata information to sqllineage to improve the lineage result.

Suppose all the tables are created in sqlite database with a file called *db.db*. In particular, table *quux* has columns *col5* and *col6* and *baz* has column *col4*.

```
sqlite3 db.db 'CREATE TABLE IF NOT EXISTS baz (bar_id int, col1 int, col4 int)';
sqlite3 db.db 'CREATE TABLE IF NOT EXISTS quux (quux_id int, col5 int, col6 int)';
```

Now given the same SQL, column lineage is fully resolved.

```
$ SQLLINEAGE_DEFAULT_SCHEMA=main sqllineage -f test.sql -l column --sqlalchemy_
↪url=sqlite:///db.db
main.corge.col1 <- main.foo.col1 <- main.bar.col1
main.corge.col2 <- main.foo.col2 <- main.bar.col1
main.corge.col2 <- main.grault.col2
main.foo.col3 <- c.col3_sum <- main.qux.col3
main.foo.col4 <- main.baz.col4
main.foo.col5 <- main.quux.col5
main.foo.col6 <- main.quux.col6
```

The default schema name in sqlite is called *main*, we have to specify here because the tables in SQL file are unqualified.

SQLLineage leverages sqlalchemy to retrieve metadata from different SQL databases. Check for more details on SQLLineage MetaData.

1.2.6 Lineage Visualization

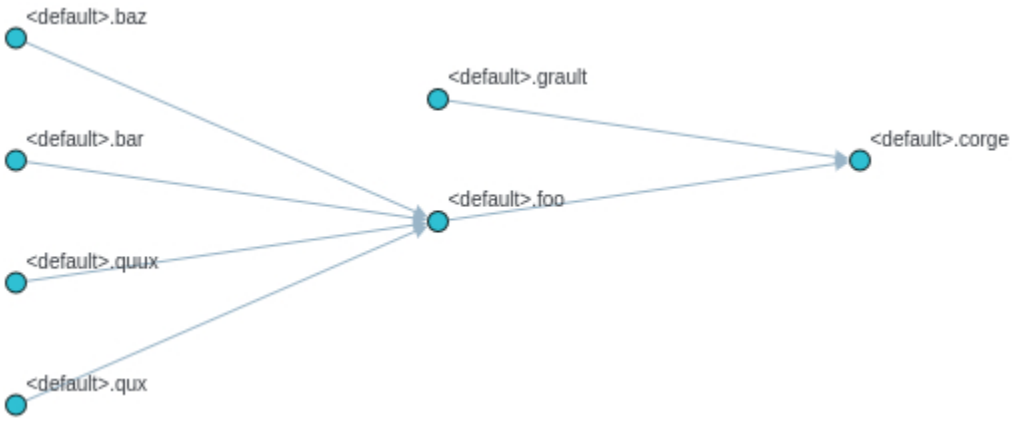
One more cool feature, if you want a graph visualization for the lineage result, toggle graph-visualization option

Still using the above SQL file:

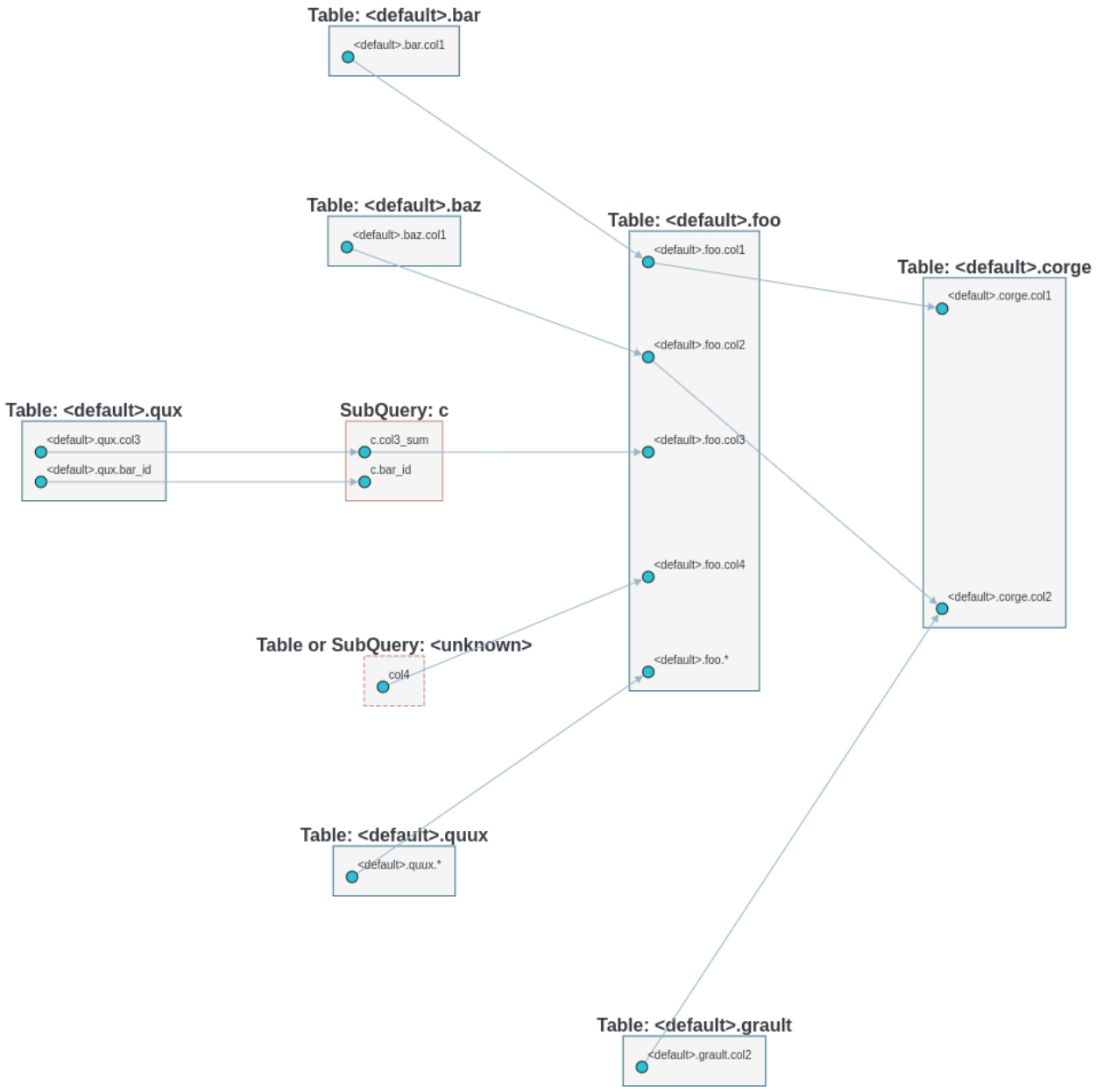
```
sqllineage -g -f test.sql
```

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)
```

(continues on next page)

(continued from previous page)

```
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()
```

Getting Started
: Install SQLLineage and quick use the handy built-in command-line tool

Advanced Usage
: Some advanced usage like multi statement SQL lineage and lineage visualization

Beyond Command Line
: Using SQLLineage in your Python script

CHAPTER

TWO

GEAR UP

2.1 Configuration

The SQLLineage configuration allows user to customize the behaviour of sqllineage.

We adopt environment variable approach for global key-value mapping. The keys listed in this section should start with *"SQLLINEAGE_"* to be a valid config. For example, to use DEFAULT_SCHEMA, use `SQLLINEAGE_DEFAULT_SCHEMA=default`.

Note: Starting v1.5.2, we also support changing config at runtime. A local config is kept for each thread that will mask global config. Local configuration must be set using context manager:

```
>>> from sqllineage.config import SQLLineageConfig
>>> from sqllineage.runner import LineageRunner

>>> with SQLLineageConfig(DEFAULT_SCHEMA="ods"):
>>>     print(LineageRunner("select * from test").source_tables)
[Table: ods.test]
>>> with SQLLineageConfig(DEFAULT_SCHEMA="dwd"):
>>>     print(LineageRunner("select * from test").source_tables)
[Table: dwd.test]
```

Note when setting local config, the key does not start with *"SQLLINEAGE_"*.

2.1.1 DEFAULT_SCHEMA

Default schema, or interchangeably called database. Tables without schema qualifier parsed from SQL is set with a schema named <*default*>, which represents an unknown schema name. If DEFAULT_SCHEMA is set, we will use this value as default schema name.

Default: ""

Since: 1.5.0

2.1.2 DIRECTORY

Frontend app SQL directory. By default the frontend app is showing the data directory packaged with sqllineage, which includes tpcds queries for demo purposes. User can customize with this key.

Default: `sqllineage/data`

Since: 1.2.1

2.1.3 LATERAL_COLUMN_ALIAS_REFERENCE

Enable lateral column alias reference. This is a syntax feature supported by some SQL dialects. See:

- Amazon Redshift: Amazon Redshift announces support for lateral column alias reference
- Spark (since 3.4): Support "lateral column alias references" to allow column aliases to be used within SELECT clauses
- Databricks: Introducing the Support of Lateral Column Alias

---

Note: Lateral column alias reference is a feature that must be used together with metadata for each column to be correctly resolved. Take below example:

```
SELECT clicks / impressions as probability,
       round(100 * probability, 1) as percentage
FROM raw_data
```

If table raw_data has a column named **probability**, **probability** in the second selected column is from table raw_data. Otherwise, it's referencing alias **clicks / impressions as probability**.

That means with SQLLineage, besides making LATERAL_COLUMN_ALIAS_REFERENCE=TRUE, MetaDataProvider must also be provided so we can query raw_data table to see if it has a column named **probability** and then check alias reference. If not provided, we will fallback to default behavior to simply assume column **probability** is from table raw_data even if LATERAL_COLUMN_ALIAS_REFERENCE is set to TRUE.

Default: `False`

Since: 1.5.1

2.1.4 TSQL_NO_SEMICOLON

Enable tsql no semicolon splitter mode. Transact-SQL offers this feature that even when SQL statements are not delimited by semicolon, it can still be parsed and executed.

> **Warning:** Quoting Transact-SQL syntax conventions (Transact-SQL), "although the semicolon isn't required for most statements in this version (v16) of SQL Server, it will be required in a future version".
>
> So with SQLLineage, this config key is kept mostly for backward-compatible purposes. We may drop the support any time without warning. Bear this in mind when using this feature.

Default: `False`

Since: 1.4.8

2.2 MetaData

Column lineage requires metadata to accurately handle case like `select *` or select unqualified columns in case of join. Without metadata, SQLLineage output partially accurate column lineage.

MetaDataProvider is a mechanism sqllineage offers so that user can optionally provide metadata information to sqllineage to improve the accuracy.

There are two MetaDataProvider implementations that sqllineage ships with. You can also build your own by extending base class *sqllineage.core.metadata_provider.MetaDataProvider*.

2.2.1 DummyMetaDataProvider

class sqllineage.core.metadata.dummy.**DummyMetaDataProvider**(*metadata: Dict[str, List[str]] | None = None*)

A Dummy MetaDataProvider that accept metadata as a dict

Parameters

metadata – a dict with schema.table name as key and a list of unqualified column name as value

By default a DummyMetaDataProvider instance constructed with an empty dict will be passed to LineageRunner. User can instantiate DummyMetaDataProvider with metadata dict of their own instead.

```
>>> from sqllineage.core.metadata.dummy import DummyMetaDataProvider
>>> from sqllineage.runner import LineageRunner
>>> sql1 = "insert into main.foo select * from main.bar"
>>> metadata = {"main.bar": ["col1", "col2"]}
>>> provider = DummyMetaDataProvider(metadata)
>>> LineageRunner(sql1, metadata_provider=provider).print_column_lineage()
main.foo.col1 <- main.bar.col1
main.foo.col2 <- main.bar.col2
>>> sql2 = "insert into main.foo select * from main.baz"
main.foo.* <- main.baz.*
```

DummyMetaDataProvider is mostly used for testing purposes. The demo above shows that when there is another SQL query like `insert into main.foo select * from main.baz`, this provider won't help because it only knows column information for table `main.bar`.

However, if somehow user can retrieve metadata for all the tables from a bulk process, then as long as memory allows, it can still be used in production.

2.2.2 SQLAlchemyMetaDataProvider

class sqllineage.core.metadata.sqlalchemy.**SQLAlchemyMetaDataProvider**(*url: str*, *engine_kwargs: Dict[str, Any] | None = None*)

SQLAlchemyMetaDataProvider queries metadata from database using SQLAlchemy

Parameters

- **url** – sqlalchemy url
- **engine_kwargs** – a dictionary of keyword arguments that will be passed to sqlalchemy create_engine

On the other hand, SQLAlchemyMetaDataProvider doesn't require user to provide metadata for all the tables needed at once. It only requires database connection information and will query the database for table metadata when needed.

```
>>> from sqllineage.core.metadata.sqlalchemy import SQLAlchemyMetaDataProvider
>>> from sqllineage.runner import LineageRunner
>>> sql1 = "insert into main.foo select * from main.bar"
>>> url = "sqlite:///db.db"
>>> provider = SQLAlchemyMetaDataProvider(url)
>>> LineageRunner(sql1, metadata_provider=provider).print_column_lineage()
```

As long as `sqlite:///db.db` is the correct source that this SQL runs on, sqllineage will generate the correct lineage.

As the name suggests, sqlalchemy is used to connect to the databases. SQLAlchemyMetaDataProvider is just a thin wrapper on sqlalchemy `engine`. SQLAlchemy is capable of connecting to multiple data sources with correct driver installed.

Please refer to SQLAlchemy Dialect documentation for connection information if you haven't used sqlalchemy before.

Note: SQLLineage only adds sqlalchemy library as dependency. All the drivers are not bundled, meaning user have to install on their own. For example, if you want to connect to snowflake using snowflake-sqlalchemy in sqllineage, then you need to run

```
pip install snowflake-sqlalchemy
```

to install the driver. After that is done, you can use snowflake sqlalchemy url like:

```
>>> use, password, account = "<your_user_login_name>", "<your_password>", "<your_account_
↪name>"
>>> provider = SQLAlchemyMetaDataProvider(f"snowflake://{user}:{password}@{account}/")
```

Make sure <your_user_login_name>, <your_password>, and <your_account_name> are replaced with the appropriate values for your Snowflake account and user.

SQLLineage will try connecting to the data source when SQLAlchemyMetaDataProvider is constructed and throws MetaDataProviderException immediately if connection fails.

Note: Some drivers allow extra connection arguments. For example, in sqlalchemy-bigquery, to specify location of your datasets, you can pass *location* to sqlalchemy `creation_engine` function:

```
>>> engine = create_engine('bigquery://project', location="asia-northeast1")
```

this translates to the following SQLAlchemyMetaDataProvider code:

```
>>> provider = SQLAlchemyMetaDataProvider('bigquery://project', engine_kwargs={"location
↪": "asia-northeast1"})
```

Configuration
: Learn how to configure sqllineage

MetaData
: Learn how to use MetaDataProvider

CHAPTER

THREE

BEHIND THE SCENE

3.1 Why SQLLineage

3.1.1 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 a software engineer, developing all kinds of 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 clause? 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/user-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.

3.1.2 Broader Use Cases

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 pains that different roles in a data team suffer from.

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 tables 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 from table flow perspective.
  - **Verify business logic**: making sure the information I use is sourcing from the correct dataset with correct transformation.

3.2 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 and apply some pre-defined rules to extract the parts 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), the database/data warehouse query engine outputs 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 the name linter is used interchangeably with code formatter. Famous tools like flake8 for Python, ESLint for JavaScript are example of real life linters. 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.

sqlfluff is the underlying parser SQLLineage uses to get the AST. You heard it right! Even though sqlfluff is mostly famous as a SQL linter, it also ships a parser so lint can be done. The various SQL dialects it supports greatly saves our time.

As mentioned, at the core of sqllineage is to traverse through the AST. Different SQL statement type requires different analyzing logic. We collect all kinds of sql, handle various edge cases and make our logic robust enough.

This is 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.

3.3 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.

3.3.1 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.

3.3.2 DONTs

- Column-level lineage will not be 100% accurate because that would require metadata information. It's optional for user to leverage MetaDataProvider functionality so sqllineage can query metadata when analyzing. If not provided, column-to-table resolution will be conducted in a best-effort way, meaning we only provide possible table candidates for situation like `select *` or `select col from tab1 join tab2`.

3.3.3 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, SQLLineage introduces an optional MetaDataProvider, where user can register metadata information in a programmatic way to assist column-to-table resolution.

3.4 Column-Level Lineage Design

3.4.1 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.

3.4.2 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: Add 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.

3.4.3 Implementation Plan

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

3.5 Dialect-Awareness Lineage Design

3.5.1 Problem Statement

As of v1.3.x release, table level lineage is perfectly production-ready. Column level lineage, under the no-metadata background, is also as good as it can be. And yet we still have a lot of corner cases that are not yet supported. This is really due to the long-tail of SQL language features and fragmentation of various SQL dialects.

Here are some typical issues:

- How to check whether syntax is valid or not?
- dialect specific syntax:
  - MSSQL assignment operator

  - Snowflake MERGE statement
  - CURRENT_TIMESTAMP: keyword or function?
  - identifier quote character: double quote or backtick?
- dialect specific keywords:
  - reversed keyword vs non-reversed keyword list
  - non-reserved keyword as table name
  - non-reserved keyword as column name
- dialect specific function:
  - Presto UNNEST
  - Snowflake GENERATOR

Over the years, we already have several monkey patches and utils on sqlparse to tweak the AST generated, either because of incorrect parsing result (e.g. parenthesized query followed by INSERT INTO table parsed as function) or not yet supported token grouping (e.g. window function for example). Due to the non-validating nature of sqlparse, that's the bitter pill to swallow when we enjoyed tons of convenience.

3.5.2 Wishful Thinking

To move forward, we'd want more from the parser so that:

1. We know better what syntax, or dialect specific feature we support.
2. We can easily revise parsing rules to generate the AST we want when we decide to support some new features.
3. User can specify the dialect when they use sqllineage, so they know what to expect. And we explicitly let them know when we don't know how to parse the SQL (InvalidSyntaxException) or how to analyze the lineage (UnsupportedStatementException).

Sample call from command line:

```
sqllineage -f test.sql --dialect=ansi
```

Sample call from Python API:

```
from sqllineage.runner import LineageRunner
sql = "select * from dual"
result = LineageRunner(sql, dialect="ansi")
```

Likewise in frontend UI, user have a dropdown select to choose the dialect they want.

3.5.3 Implementation Plan

OpenMetadata community contributed an implementation using the parser underneath sqlfluff. With #326 merged into master, we have a new *dialect* option. When passed with real dialect, like mysql, oracle, hive, sparksql, bigquery, snowflake, etc, we'll leverage sqlfluff to analyze the query. A pseudo dialect *non-validating* is introduced to remain backward compatibility, falling back to use sqlparse as parser.

We're running dual test using both parser and make sure the lineage result is exactly the same for every test case (except for a few edge cases).

From code structure perspective, we refactored the whole code base to introduce a parser interface:

- LineageAnalyzer now accepts single statement SQL string, split by LineageRunner, and returns StatementLineageHolder as before
- Each parser implementations sit in folder **sqllineage.core.parser**. They're extending the LineageAnalyzer, common Models, and leverage Holders at different layers.

Note: Dialect-awareness lineage is now released with v1.4.0

Why SQLLineage
The motivation of writing SQLLineage

How Does SQLLineage Work
The inner mechanism of SQLLineage

DOs and DONTs
Design principles for SQLLineage

Column-Level Lineage Design
Design docs for column lineage

Dialect-Awareness Lineage Design
Design docs for dialect-awareness lineage

CHAPTER

FOUR

BASIC CONCEPTS

4.1 Runner

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 `sqllineage.utils.helpers.split` function to split string-base SQL statements into a list of `str` statements.
2. Calling *sqllineage.core.analyzer.LineageAnalyzer* to analyze each one statement sql string. Get 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.

4.1.1 sqllineage.runner.LineageRunner

class sqllineage.runner.**LineageRunner**(*sql: str*, *dialect: str = 'ansi'*, *metadata_provider: ~sqllineage.core.metadata_provider.MetaDataProvider = <sqllineage.core.metadata.dummy.DummyMetaDataProvider object>*, *verbose: bool = False*, *silent_mode: bool = False*, *draw_options: ~typing.Dict[str, str] | None = None*)

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

Parameters

- **sql** – a string representation of SQL statements.
- **dialect** – sql dialect
- **metadata_provider** – metadata service object providing table schema
- **verbose** – verbose flag indicating whether statement-wise lineage result will be shown
- **silent_mode** – boolean flag indicating whether to skip lineage analysis for unknown statement types

__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

static supported_dialects() → Dict[str, List[str]]

an ordered dict (so we can make sure the default parser implementation comes first) with key, value as parser_name, dialect list respectively

4.1.2 sqllineage.cli.main

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

The command line interface entry point.

Parameters

args – the command line arguments for sqllineage command

4.2 Analyzer

LineageAnalyzer is an abstract class, supposed to include the core processing logic for one-statement SQL analysis.

Each parser implementation will inherit LineageAnalyzer and do parser specific analysis based on the AST they generates and store the result in `StatementLineageHolder`.

4.2.1 LineageAnalyzer

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

SQL Statement Level Lineage Analyzer Parser specific implementation should inherit this class and implement analyze method

abstract analyze(*sql: str*, *metadata_provider:* MetaDataProvider) → *StatementLineageHolder*

to analyze single statement sql and store the result into StatementLineageHolder.

Parameters

- **sql** – single-statement SQL string to be processed
- **metadata_provider** – *sqllineage.core.metadata_provider.MetaDataProvider* provides metadata on tables to help lineage analyzing

Returns

sqllineage.core.holders.StatementLineageHolder

4.3 Holder

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

At the bottom, we have *sqllineage.core.holders.SubQueryLineageHolder* to hold lineage at subquery level. This is used internally by *sqllineage.core.analyzer.LineageAnalyzer*.

LineageAnalyzer generates `sqllineage.core.holder.StatementLineageHolder` as the result of lineage at SQL statement level.

To assemble multiple `sqllineage.core.holder.StatementLineageHolder` into a DAG based data structure serving for the final output, we have *sqllineage.core.holders.SQLLineageHolder*

4.3.1 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.core.models.Table*].

This is the most atomic representation of lineage result.

property write_columns: List[*Column*]

return a list of columns that write table contains. It's either manually added via *add_write_column* if specified in DML or automatic added via *add_column_lineage* after parsing from SELECT

add_write_column(**tgt_cols:* Column) → None

in case of DML with column specified, like:

```
INSERT INTO tab1 (col1, col2)
SELECT col3, col4
```

this method is called to make sure tab1 has column col1 and col2 instead of col3 and col4

add_column_lineage(*src:* Column, *tgt:* Column) → None

link source column to target.

get_alias_mapping_from_table_group(*table_group: List[Path* | Table | SubQuery*]*) → Dict[str, Path | *Table* | *SubQuery*]

A table can be referred to as alias, table name, or database_name.table_name, create the mapping here. For SubQuery, it's only alias then.

4.3.2 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.core.models.Table*].

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

4.3.3 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.core.models.Table*

property target_tables: Set[*Table*]

a list of target *sqllineage.core.models.Table*

property intermediate_tables: Set[*Table*]

a list of intermediate *sqllineage.core.models.Table*

static of(*metadata_provider*, **args:* StatementLineageHolder) → *SQLLineageHolder*

To assemble multiple *sqllineage.core.holders.StatementLineageHolder* into *sqllineage.core.holders.SQLLineageHolder*

4.4 Model

Several data classes in this module.

4.4.1 Schema

class sqllineage.core.models.**Schema**(*name: str | None = None*)

Data Class for Schema

Parameters
name – schema name

4.4.2 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*

4.4.3 SubQuery

class sqllineage.core.models.**SubQuery**(*subquery: Any*, *subquery_raw: str*, *alias: str | None*)

Data Class for SubQuery

Parameters

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

4.4.4 Column

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

Data Class for Column

Parameters

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

4.5 MetaDataProvider

4.5.1 sqllineage.core.metadata_provider.MetaDataProvider

class sqllineage.core.metadata_provider.**MetaDataProvider**

Base class used to provide metadata like table schema.

When parse below sql:

```
INSERT INTO db1.table1
SELECT c1
FROM db2.table2 t2
JOIN db3.table3 t3 ON t2.id = t3.id
```

Only by literal analysis, we don't know which table is selected column c1 from. A subclass of MetaDataProvider implementing _get_table_columns passing to *sqllineage.runner.LineageRunner*. can help parse column lineage correctly.

get_table_columns(*table:* Table, ***kwargs*) → List[*Column*]

return columns of given table.

register_session_metadata(*table:* Table, *columns: List[*Column*]*) → None

Register session-level metadata, like temporary table or view created.

deregister_session_metadata() → None

Deregister session-level metadata.

Runner

LineageRunner: The entry point for SQLLineage

Analyzer

LineageAnalyzer: The core functionality of analyze one SQL statement

Holder
: LineageHolder: To hold lineage result at different level

Model
: The data classes for SQLLineage

MetaDataProvider
: MetaDataProvider: provider metadata to assist lineage analysis

CHAPTER

FIVE

RELEASE NOTE

5.1 Changelog

5.1.1 v1.5.2

Date
April 7, 2024

Enhancement

- Enable support of sqlfluff context (#548)
- Support Change Configure Default Schema At Runtime (#536)

Bugfix

- Parse column level lineage incorrect (#584)
- Metadata Masked When Table was in a previous UPDATE statement (#577)
- Clickhouse SQL 'GLOBAL IN' not support (#554)
- Support json_tuple in SELECT clause in Hive (#483)

5.1.2 v1.5.1

Date
February 4, 2024

This is a bugfix release mostly driven by community contributors. Thanks everyone for making SQLLineage better.

Enhancement

- Allow unambiguous column reference for JOIN with USING clause (#558)
- Make Lateral Column Alias Reference Configurable (#539)
- Add an Argument to Exclude SubQuery Column Node in Column Lineage Path (#526)

Bugfix

- Not fully processed top-level subquery in DML (#564)
- Missing target table with tsql parsing into statements with union (#562)
- The second and subsequent case when subqueries in the select_clause are not correctly recognized (#559)
- SQLLineageConfig boolean value returns True for all non-empty strings (#551)
- Column lineage does not traverse through CTE containing uppercase letters (#531)

5.1.3 v1.5.0

Date
January 7, 2024

Great thanks to liznzn for contributing on MetaData-awareness lineage. Now we're able to generate more accurate column lineage result for *select* * or select unqualified columns in case of table join through a unified MetaDataProvider interface.

Also a breaking change is made to make ansi the default dialect in v1.5.x release as we target ultimately deprecating non-validating dialect in v1.6.x release.

Breaking Change

- Make ansi the Default Dialect (#518)

Feature

- Metadata Provider to Assist Column Lineage Analysis (#477)

Enhancement

- Add a Configuration for Default Schema (#523)
- Silent Mode Option to Suppress UnsupportedStatementException (#513)
- Support Lateral Column Alias Reference Analyzing (#507)
- Skip Lineage Analysis for SparkSQL Function Related Statement (#500)
- update statement column lineage (#487)

Bugfix

- subquery mistake alias as table name in visualization (#512)
- InvalidSyntaxException When SQL Statement Ends with Multiple Semicolons (#502)
- Misidentify Binary Operator * As Wildcard (#485)
- adding type cast operator produces different results for redshift dialect (#455)

5.1.4 v1.4.9

Date
December 10, 2023

This is a bugfix release where we closed a bunch of issues concerning CTE and UNION

Bugfix

- Not Using Column Name Specified in Query For CTE within Query (#486)
- CTE (Common Table Expressions) within CTE (#484)
- lineage inaccurate when CTE used in subquery (#476)
- UNION ALL Queries resolves column lineage incorrectly (#475)
- Missing table when parsing sql with UNION ALL (#466)
- No target tables in UPDATE statement using CTE (#453)

5.1.5 v1.4.8

Date
October 16, 2023

Enhancement

- Support Python 3.12 (#469)
- programmatically list supported dialects (#462)
- add versioning of package to cli (#457)
- Add Support of DROP VIEW statements (#456)
- support split SQL statements without semicolon in tsql (#384)

Bugfix

- SqlFluff RuntimeError Triggers Server Error 500 in Frontend (#467)
- ignore lineage for analyze statement (#459)

5.1.6 v1.4.7

Date
August 27, 2023

Enhancement

- Support subquery in VALUES clause (#432)
- Dialect='tsql' should return warning when no semicolons are detected (#422)
- Restricting folder and files user can access from frontend (#405)
- throw exception when the statement missing the semicolon as splitter (#159)

Bugfix

- AttributeError raised using parenthesized where clause (#426)
- qualified wildcard recognized as wrong column name (#423)

5.1.7 v1.4.6

Date
July 31, 2023

In this release, we finally reach the milestone to make all sqlparse only test cases passed with sqlfluff implementation. That's a big step in ultimately deprecating sqlparse. Also by upgrading to latest version of sqlfluff (with our PR merged), we enjoy the benefits of improved sqlfluff performance when parsing some SQLs with nested query pattern.

Enhancement

- Improve sqlfluff Performance Issue on Nested Query Pattern (#348)
- Reduce sqlparse only test cases (#347)

Bugfix

- Missing Source Table for MERGE statement when UNION involved in source subquery (#406)
- Column lineage does not work for CAST to Parameterized Data Type (#329)
- Can't handle parenthesized from clause (#278)

5.1.8 v1.4.5

Date
July 2, 2023

Enhancement

- Switch to PyPI Trusted Publishers (#389)
- Support tsql Declare Statement (#357)

Bugfix

- Exception for Subquery Expression Without Source Tables (#401)
- Not Supporting Create Table AS in postgres (#400)
- Failed to handle UNION followed by CTE (#398)
- Not handling CTE inside DML query (#377)
- Failed to parse UNION inside CTE (#376)

5.1.9 v1.4.4

Date
June 11, 2023

Enhancement

- BigQuery Specific MERGE statement feature support (#380)
- Support snowflake create table…clone and alter table…swap (#373)
- Parse Column Lineage When Specify Column Names in Insert/Create Statement (#212)

Bugfix

- Switching Dialect in UI only works When Explicit Clicked (#387)
- No Column Lineage Parsed for DML with SELECT query in parenthesis (#244)

5.1.10 v1.4.3

Date
May 13, 2023

Enhancement

- Support postgres style type casts "keyword::TIMESTAMP" (#364)

Bugfix

- Missing column lineage from SELECT DISTINCT using non-validating dialect (#356)
- Missing column lineage with Parenthesis around column arithmetic operation (#355)
- Not Handling CTE at the start of query in DML (#328)

5.1.11 v1.4.2

Date
April 22, 2023

Bugfix

- sqlparse v0.4.4 breaks non-validating dialect (#361)

5.1.12 v1.4.1

Date
April 2, 2023

Bugfix

- frontend app unable to load dialect when launched for the first time

5.1.13 v1.4.0

Date
March 31, 2023

Great thanks to Nahuel, Mayur and Pere from OpenMetadata community for contributing on feature Dialect-awareness lineage. Leveraging sqlfluff underneath, we're now able to give more correct lineage result with user input on SQL dialect.

Feature

- Dialect-awareness lineage (#302)
- support MERGE statement (#166)

Enhancement

- Use curved lines in lineage graph visualization (#320)
- Click to lock highlighted nodes in visualization (#318)
- Deprecate support for Python 3.6 and Python 3.7, add support for Python 3.11 (#319)
- support t-sql assignment operator (#205)

Bugfix

- exception when insert into qualified table followed by parenthesized query (#249)
- missing columns when current_timestamp as reserved keyword used in select clause (#248)
- exception when non-reserved keywords used as column name (#183)
- exception when non-reserved keywords used as table name (#93)

5.1.14 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)

5.1.15 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)

5.1.16 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)

5.1.17 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)

5.1.18 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)

5.1.19 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)

5.1.20 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)

5.1.21 v1.3.0

Date
November 13, 2021

Feature

- Column-Level Lineage (#103)

Bugfix

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

5.1.22 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)

5.1.23 v1.2.3

Date
May 15, 2021

Enhancement

- lineage API response exception handling (#148)

5.1.24 v1.2.2

Date
May 5, 2021

Bugfix

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

5.1.25 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)

5.1.26 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)

5.1.27 v1.1.4

Date
March 9, 2021

Bugfix

- trim function with from in arguments (#127)

5.1.28 v1.1.3

Date
February 1, 2021

Bugfix

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

5.1.29 v1.1.2

Date
January 26, 2021

Bugfix

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

5.1.30 v1.1.1

Date
January 24, 2021

Bugfix

- SQLLineageException for Multiple CTE Subclauses (#115)

5.1.31 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)

5.1.32 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)

5.1.33 v1.0.1

Date
October 17, 2020

Enhancement

- remove upper bound for dependencies (#85)

5.1.34 v1.0.0

Date
September 27, 2020

New Features

- a detailed documentation hosted by readthedocs (#81)

Enhancement

- drop support for Python 3.5 (#79)

5.1.35 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)

5.1.36 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)

5.1.37 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)

5.1.38 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)

5.1.39 v0.0.1

Date
June 16, 2019

New Features

initial public release

Changelog
See what's new for each SQLLineage version

INDEX

Symbols

__str__() (*sqllineage.runner.LineageRunner method*), 23

A

add_column_lineage() (*sqllineage.core.holders.SubQueryLineageHolder method*), 25
add_write_column() (*sqllineage.core.holders.SubQueryLineageHolder method*), 25
analyze() (*sqllineage.core.analyzer.LineageAnalyzer method*), 24

C

Column (*class in sqllineage.core.models*), 27
column_lineage_graph (*sqllineage.core.holders.SQLLineageHolder property*), 26

D

deregister_session_metadata() (*sqllineage.core.metadata_provider.MetaDataProvider method*), 27
draw() (*sqllineage.runner.LineageRunner method*), 23
DummyMetaDataProvider (*class in sqllineage.core.metadata.dummy*), 13

G

get_alias_mapping_from_table_group() (*sqllineage.core.holders.SubQueryLineageHolder method*), 25
get_table_columns() (*sqllineage.core.metadata_provider.MetaDataProvider method*), 27

I

intermediate_tables (*sqllineage.core.holders.SQLLineageHolder property*), 26

L

LineageAnalyzer (*class in sqllineage.core.analyzer*), 24
LineageRunner (*class in sqllineage.runner*), 23

M

main() (*in module sqllineage.cli*), 24
MetaDataProvider (*class in sqllineage.core.metadata_provider*), 27

O

of() (*sqllineage.core.holders.SQLLineageHolder static method*), 26

P

print_column_lineage() (*sqllineage.runner.LineageRunner method*), 24
print_table_lineage() (*sqllineage.runner.LineageRunner method*), 24

R

register_session_metadata() (*sqllineage.core.metadata_provider.MetaDataProvider method*), 27

S

Schema (*class in sqllineage.core.models*), 26
source_tables (*sqllineage.core.holders.SQLLineageHolder property*), 26
SQLAlchemyMetaDataProvider (*class in sqllineage.core.metadata.sqlalchemy*), 13
SQLLineageHolder (*class in sqllineage.core.holders*), 26
StatementLineageHolder (*class in sqllineage.core.holders*), 25
SubQuery (*class in sqllineage.core.models*), 27
SubQueryLineageHolder (*class in sqllineage.core.holders*), 25
supported_dialects() (*sqllineage.runner.LineageRunner static method*), 24

T

Table (*class in sqllineage.core.models*), 26
table_lineage_graph (*sqllineage.core.holders.SQLLineageHolder property*), 26
target_tables (*sqllineage.core.holders.SQLLineageHolder property*), 26

W

write_columns (*sqllineage.core.holders.SubQueryLineageHolder property*), 25