LINEAGEX: A Column Lineage Extraction System for SQL

Shi Heng Zhang
Simon Fraser University
Burnaby, Canada
andy_zhang@sfu.ca

Zhengjie Miao
Simon Fraser University
Burnaby, Canada
zhengjie@sfu.ca

Jiannan Wang
Simon Fraser University
Burnaby, Canada
jnwang@sfu.ca

***Abstract*—As enterprise data grows in size and complexity, column-level data lineage, which records the creation, transformation, and reference of each column in the warehouse, has been the key to effective data governance that assists tasks like data quality monitoring, storage refactoring, and workflow migration. Unfortunately, existing systems introduce overheads by integration with query execution or fail to achieve satisfying accuracy for column lineage. In this paper, we demonstrate LINEAGEX, a lightweight Python library that infers column-level lineage from SQL queries and visualizes it through an interactive interface. LINEAGEX achieves high coverage and accuracy for column lineage extraction by intelligently traversing query parse trees and handling ambiguities. The demonstration walks through use cases of building lineage graphs and troubleshooting data quality issues. LINEAGEX is open sourced at https://github.com/sfu-db/lineagex and our video demonstration is at https://youtu.be/5LaBBDDitlw**

***Index Terms*—database, lineage, provenance**

I. INTRODUCTION

Data governance has become increasingly crucial as data is becoming larger and more complex in enterprise data warehouses. For example, in an organization's data pipeline, data flows from upstream artifacts to downstream services, which may be built by various teams that know little about other teams' work and often introduce challenges when anyone wants to change their data. In this case, lineage [9], [10], especially finer-grained *column-level lineage*, is often needed for simplifying the impact analysis of such a change, i.e., how a change in the upstream would affect the downstream. In another real-world scenario, column-level lineage can help identify how sensitive data flows throughout the entire pipeline, thereby improving the overall data quality and validating data compliance with regulations, such as GDPR and HIPAA [7].

While capturing lineage information in DBMS has been studied extensively in the database community [1], [2], [11], the need remains to curate the lineage information from static analysis of queries (without executing the queries). On the one hand, existing systems or tools would introduce large overheads by either modifying the database internals [1], [2] or rewriting the queries to store the lineage information [11], [12]. On the other hand, different data warehouse users may need to disaggregate the lineage extraction workflow from query execution to simplify their collaboration, as shown in the following example.

Fig. 1. Lineage extraction from query logs without a database connection.

Example 1: An online shop uses a data warehouse to store and analyze its customer and transaction data. There is a view, `webinfo`, which keeps track of user activities, and another view, `info`, connects the users' website activities (stored in view `weact`) to their orders, which may be used for recommendation purposes. However, the online shop owner decides to edit the `page` column of the `web` table and requests an impact analysis from the data warehouse provider.

```
Q1 = CREATE VIEW info AS
SELECT c.name, c.age, o.oid, w.*
FROM customers c JOIN orders o ON c.cid = o.cid
JOIN weact w ON c.cid = w.wcid;
Q2 = CREATE VIEW weact AS
SELECT w.wcid, w.wdate, w.wpage, w.wreg
FROM webinfo w
INTERSECT
SELECT w1.cid, w1.date, w1.page, w1.reg
FROM web w1;
Q3 = CREATE VIEW webinfo AS
SELECT c.cid AS wcid, w.date AS wdate,
  w.page AS wpage, w.reg AS wreg
FROM customers c JOIN web w ON c.cid = w.cid
WHERE EXTRACT(YEAR from w.date) = 2022;
```

Due to access control and privacy regulations, the engineer from the data warehouse provider can only access the log of database queries instead of the DBMS. The task is prone to being time-consuming and may involve tracing unnecessary columns without a comprehensive data flow overview. To address this, the engineer considers using tools like SQLLineage [6] to extract and visualize the lineage graph.

Although it can generate a lineage graph as shown in Figure 2, there are a few issues with the column lineage. One is that the node of `weact` erroneously includes four extra columns, highlighted in a solid red rectangle. Another error arises for view `info` due to the `SELECT *` operation, which makes it unable to match the output columns to columns in `weact`. Instead, it would return an erroneous entry of `weact.*` to `info.*` (in solid red rectangle) while omitting

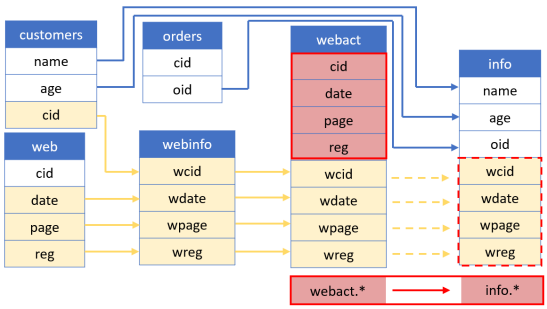

Fig. 2. The lineage graph for Example 1. Existing tools like SQLLineage [6] would miss columns in the dashed red rectangle and return wrong entries in the solid red rectangle, while the yellow is the correct lineage

the four correct columns from `webact`. It would also return fewer columns for the view `info` (in dashed red rectangle) and completely ignore the edges connecting `webact` to it (the yellow dashed arrows). If the engineer used the information from this lineage graph, then not only an erroneous column (`webact.page`) is provided, but the results also miss actual impacted columns from the `webact` and `info` table. As we will demonstrate, our approach is able to handle statements like `SELECT w.*` and capture all columns and their dependencies missed by prior tools.

Curating lineage information from query logs is also advantageous for debugging data quality issues, enhancing data governance, refactoring data, and providing impact analysis. However, existing tools [5], [6] often fail to accurately infer column lineage due to the absence of metadata. To support developers and analysts in extracting lineage without the overhead of running queries in DBMS, we develop a lightweight Python library, LINEAGEX, which *constructs a column-level lineage graph from the set of query definitions and provides concise visualizations of how data flows in the DBMS.*

Challenges. LINEAGEX achieves accurate column-level lineage extraction by addressing the following two challenges. First is the *variety of SQL features*, especially for features that involve intermediate results or introduce column ambiguity. For example, Common Table Expressions (CTEs) and subqueries generate intermediate results that the output columns depend on, while the desired lineage should only reveal the source tables and columns. Set operations may introduce column ambiguity, primarily due to the lack of table prefixes. Second, when there is an *absence of metadata* from the DBMS on each table's columns, e.g., when the query uses `SELECT *` or refers to a column without its table prefix, it may introduce ambiguities. Thus, prior works fail to trace the output columns when the $\star$ symbol exists and cannot identify their original table without an explicit table prefix.

Our contributions. For the first challenge, LINEAGEX uses a SQL parser to obtain the queries' abstract syntax trees (AST) and perform an *intelligently designed traversal on the AST with a comprehensive set of rules to identify column dependencies*. LINEAGEX addresses the second challenge by *dynamically adjusting the processing order for queries* when it identifies ambiguities in the source of tables or columns. Moreover, to accommodate the majority of data practitioners, we integrate

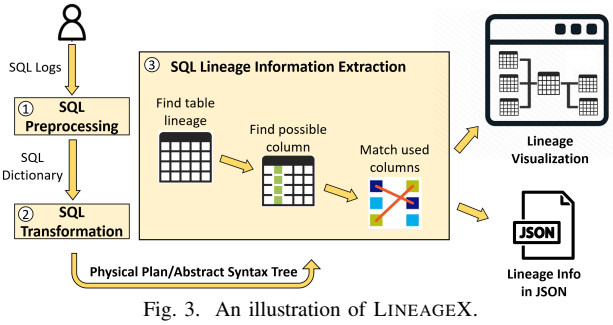

Fig. 3. An illustration of LINEAGEX.

LINEAGEX with the popular Python data science ecosystem by providing a *simple API* that directly takes the SQL statements and outputs the lineage graph. Besides the API, we provide a UI that *visualizes the column lineage* for users to examine.

In this demonstration, we will showcase the impact analysis scenario and illustrate how LINEAGEX provides accurate column-level lineage to further help users monitor their data flow. The user can compare the lineage extraction results by LINEAGEX with prior tools. Since pre-trained large language models (LLMs) have shown impressive performance in understanding code, we will also demonstrate using state-of-the-art LLMs like GPT-4o for impact analysis and how to augment their results with the column-level lineage from LINEAGEX.

II. BACKGROUND AND RELATED WORK

Data lineage tracks how data flows between each step in a data processing pipeline. Consider each processing step as a query Q, the *table-level lineage* $\mathtt{T}$ of Q encodes which input tables contribute to its output; and the *column-level lineage* $\mathtt{C}$ is a mapping from Q's output columns $\mathcal{C}^{output}$ to Q's input columns $\mathcal{C}^{source}$, which encodes for each output column which specific columns in the input tables it relies on. More specifically, for an output column $c^{out} \in \mathcal{C}^{output}$ of Q, an input column $C^{src} \in \mathcal{C}^{source}$ is included in $\mathtt{C}(c^{out})$ if any changes to C^{src} will lead to a potential change in the values in c^{out} — we may not only include the input columns directly contributing to the output value but also take any column referred in the query into consideration.

Then, consider a set of queries $\mathcal{Q} = \{Q_i\}$, *lineage extraction* is to find the pair $(\mathtt{T}_i, \mathtt{C}_i)$ for each Q_i. Note that queries in $\mathcal{Q}$ may be table/view creation queries, hence $\mathtt{T}_i$ and $\mathtt{C}_i$ may map the outputs of Q_i to the outputs of other queries. In practice, to make the lineage graph easy to read, we can combine these two graphs and group all columns' output by the same query to visualize this graph.

Related work. Data lineage [9], [10] has been studied extensively in the database research community. To track fine-grained lineage information down to the tuple level or cell level, people have extended relational database engines like in ProvSQL [1] and PERM [2] or built middlewares that rewrite queries [11], [12], which are often "overkill" for column-level lineage. Various industry-leading tools, including Linkedin's Datahub [8], Microsoft's Purview [4], and Apache Atlas [3], are more than capable of handling data pipelines and relational databases, but they may incur high operational and

TABLE I
KEYWORD RULES.

Keyword	Process	Explanation
SELECT	$C^{con} \leftarrow p \cup C^{pos} \forall p \in \mathcal{P}$	Resolve C^{con} for each projection
FROM (Table/ View)	$T \leftarrow T \cup \{\textit{this table}\}$ $C^{pos} \leftarrow C^{pos} \cup \{\textit{its columns}\}$	Add to T for table lineage, and add its column to C^{pos}
FROM (CTE/ Subquery)	*find the CTE/Subquery in* M_{CTE} $C^{pos} \leftarrow C^{pos} \cup \{\textit{its columns}\}$	Find this CTE/subquery in M_{CTE}, and add its columns to C^{pos}
WITH/ Subquery	$M_{CTE} \leftarrow T, C^{con}, C^{ref}$ $T, C^{con}, C^{ref}, C^{pos}, \mathcal{P} \leftarrow \emptyset$	M_{CTE} gets all the current table and column lineage, store to be referenced
Set Operation	$C^{ref} \leftarrow C^{ref} \cup p \cup C^{pos}\ \forall p \in \mathcal{P}$ $C^{pos}, \mathcal{P} \leftarrow \emptyset$ *repeat for other leaves*	Add all the columns in the projection to C^{ref}, this process is repeated for other leaves connected
Other Keywords	$temp_{cols} \leftarrow$ *all columns here* $C^{ref} \leftarrow C^{ref} \cup p \cup C^{pos}$ $\forall p \in temp_{cols}$	Add all the columns found here to C^{ref}

maintenance costs. Vamsa [13] annotates columns used to train machine learning models for Python scripts. There are also Python libraries like SQLGlot [5] and SQLLineage [6] that parse SQL queries statically; however, they focus on lineage for individual files, lacking the ability to find the dependency across queries, especially when there are ambiguities in table or column names. None of the methods above provides lightweight and accurate lineage extraction at the column level, like what LINEAGEX offers, without running the database queries; LINEAGEX can also visualize related tables and the data flow between columns in an interactive graph.

III. SYSTEM AND IMPLEMENTATION

The overview of the LINEAGEX system is shown in Figure 3. LINEAGEX allows users to input a list of SQL statements or query logs. Below are details of each module.

SQL Preprocessing Module. The first step is to scan each query and record the mappings from the query's identifier to its query body. For `CREATE` statements, we use the created table/view's name as the query identifier, while for `SELECT` statement-only queries, we use a randomly generated id[1]. Then, each identifier is mapped to the body of the `SELECT` statement, forming a key-value pair. For instance, for Q_3 in Example 1, our module would have `webinfo` as the key and the `SELECT` statement ... (line 12 to 15) as the value. These key-value pairs are stored in a Query Dictionary (QD), which will be further used to facilitate the inference between queries and identify the query dependencies.

SQL Transformation Module. Then, the Transformation Module reads each entry in the dictionary QD from the Preprocessing Module, generating an abstract syntax tree (AST) using a SQL parser (in the implementation, we used SQLGlot). The SQL AST captures all keywords and expressions in the query in a tree-like format, where the leaf nodes represent the initial scanning of source tables or the parameters of each operator, the root represents the final step, and intermediate nodes represent relational operators in the query.

[1]For some systems like dbt, queries containing only `SELECT` statement are stored in separate files. In this case, we will use the file name as the query identifier. We also provide a dbt-specific wrapper for LINEAGEX.

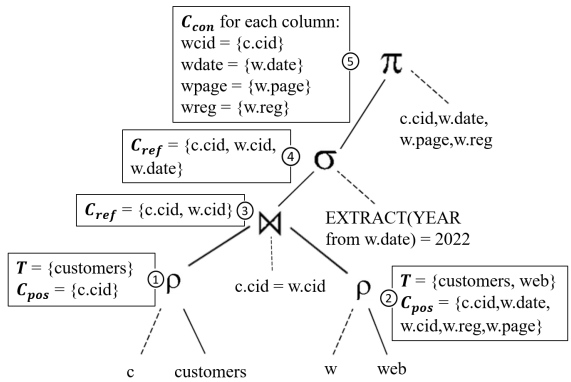

Fig. 4. Sample AST and traverse order

SQL Lineage Information Extraction Module. The final module takes each query AST as input and builds the mappings from the result view/table to its lineage `T` and the mapping from output columns C^{output} to input columns C^{source}. We consider three types of columns in the lineage: 1)C^{con}: columns that directly contribute to C^{output}; 2) C^{ref}: columns referenced in the query, e.g., columns used in the join predicate or the `WHERE` clause; and 3) C^{both}: columns in both C^{con} and C^{ref}. The extraction process involves traversing the AST with a post-order Depth-First Search (DFS), for which we create some temporary variables: M_{CTE} is a mapping for the table and column lineage information from `WITH/subquery`, C^{pos} denotes column candidates, and $\mathcal{P}$ denotes the resulting columns of the most recent projection. When encountering different keywords, the lineage information and temporary variables will be updated according to the rules in Table I.

An example for traversing the AST of Q_3 is shown in Figure 4. ①: The traversal starts with the leaf node, scanning of `customers`, so it follows the `FROM` Rule by adding it to `T` and its columns to C^{pos}. ②: The next node is scanning of `web`, so it is added to `T` and its columns to C^{pos}. ③: The next node is a `JOIN`, following the `Other keywords` Rule: `customers.cid` and `web.cid` are added to to C^{ref}. ④: For the `WHERE` node (σ), same rule applies, hence adding `web.date` to C^{ref}. ⑤: The last node is the `SELECT` (π), applying the `SELECT` Rule. Each output column's C^{con} only has one column, e.g., `wcid` has C^{con} of `customers.cid`.

Table/View Auto-Inference. In the Lineage Information Extraction module, the system gives priority to SQL statements identified by keys in QD from the Preprocessing Module. This procedure leverages a stack to reorder the query ASTs to traverse, where current traversal is temporarily deferred and placed onto the stack. That is, in cases where the tables or views encountered during the traversal have not been processed yet, they are pushed to the stack. Once the lineage information of missing tables is extracted, the deferred operation is popped from the stack following a Last-In-First-Out protocol and resumes. This strategic approach plays a pivotal role in handling `SELECT` $\star$ statements and resolving ambiguities related to columns without a prefixed table name.

When the database connection is available. While primarily focusing on *static* lineage extraction from query logs, LIN-

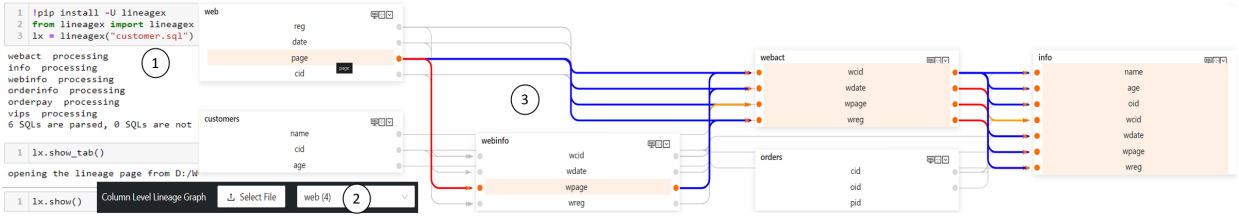

Fig. 5. The User Interface of LINEAGEX.

EAGEX can also incorporate the extraction with a database connection. We extended LINEAGEX using PostgreSQL's `EXPLAIN` command to obtain the physical query plan instead of the AST from the parser, which provides accurate metadata to deal with table and column reference ambiguities. Similar to the absent views or tables in the static extraction, an error may occur due to missing dependencies when running the `EXPLAIN` command. This requires the stack mechanism and performing an additional step to create the views first to ensure the presence of the necessary dependencies.

IV. DEMONSTRATION

We will walk through the audience with the use cases like Example 1, employing multiple datasets, such as the MIMIC dataset [2] in the healthcare domain. The MIMIC dataset has a reasonably complex schema with more than 300 columns in 26 base tables and 700 columns in 70 view definitions. We demonstrate in detail each step of using LINEAGEX for our running example in the environment of a Jupyter Notebook.

Step 1: Get started. Users have the flexibility to store their SQL queries in either files or a Python list. In this example, all SQL queries are stored in the file `customer.sql`. Then the function call is straightforward, as outlined in Figure 5 ①, the users simply install and import the library, then call the LINEAGEX function. The result will be returned in a JSON file (lineage information) and an HTML file (lineage graph).

Step 2: Locating the table. Next, users can visualize the graph using the `show` function in the notebook or the `show_tab` to open a webpage. Moreover, users can select the table of interest through a dropdown menu, as shown in Figure 5 ②. Subsequently, the target table `web` and its corresponding columns are displayed.

Step 3: Navigating column dependency. Users can click the `explore` button on the top right of the table to reveal the table's upstream and downstream tables, presenting the initial table lineage. The data flows from left to right on the visualization — tables on the right are dependent on tables on the left. Since we are doing an impact analysis, that is to find all the downstream columns and their downstream columns and so on. The first `explore` action would only show `webinfo` and `weback` tables, since they are the only ones that are directly dependent on the `web` table. The next `explore` action would reveal the `info` table, and there would be no more downstreams for `info`. With the lineage graph, hovering over the `page` column highlights all of its downstream columns, as shown in Figure 5 ③.

[2]https://github.com/MIT-LCP/mimic-code

Step 4: Solving the case. The `page` column directly contributes to `wpage` from `webinfo`(shown in red), so it is definitely impacted. The `webact` table is a result of a set operation from `web` and `webinfo`, therefore all of the `webact`'s columns will reference the `page` column and thus all get impacted(shown in blue and orange when it is both referenced and contributed). Since the `wcid` column is impacted, it is also in the `JOIN` operation for the `info` table, then all of the columns would reference the `wcid` column and potentially get impacted. Therefore, the end result for the impact analysis would be `webinfo.wpage` and all of the columns from the `webact` and `info` tables.

Comparison with existing methods. In our demonstration, users can compare results from LINEAGEX with those from SQLLineage [6]. SQLLineage returns incorrect columns for `info` and lacks lineage information for columns derived from `webinfo`, as shown in Figure 2. The users can also see how state-of-the-art LLMs respond to their questions about impact analysis: for example, GPT-4o is able to correctly identify all contributing columns impacted by changes to `page`—specifically, the `wpage` columns in `webinfo`, `weback`, and `info` tables (highlighted in red or orange), but it is not able to reveal the columns that are referenced (not directly contributing to) in the SQL (such as the `weback.wcid` in the `JOIN` condition).

REFERENCES

[1] P. Senellart *et al.*, "ProvSQL: Provenance and Probability Management in PostgreSQL," *PVLDB*, vol. 11, no. 12, pp. 2034–2037, 2018.

[2] B. Glavic and G. Alonso, "Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting," in *Proc. ICDE*, Shanghai, China, Mar. 29 - Apr. 2, 2009, pp. 174–185.

[3] M. Tang *et al.*, "SAC: A System for Big Data Lineage Tracking," in *Proc. ICDE*, Macao, China, Apr. 8-11, 2019, pp. 1964–1967.

[4] S. Ahmad *et al.*, "Microsoft Purview: A System for Central Governance of Data," *PVLDB*, vol. 16, no. 12, pp. 3624–3635, 2023.

[5] T. Mao, "sqlglot," GitHub repository, 2024, [Online]. Available: https://github.com/tobymao/sqlglot.

[6] J. Hu, "sqllineage," GitHub repository, 2024, [Online]. Available: https://github.com/reata/sqllineage.

[7] C. Dai *et al.*, "An Approach to Evaluate Data Trustworthiness Based on Data Provenance," in *Secure Data Management*, Berlin, Heidelberg, 2008, pp. 82–98.

[8] A. P. Bhardwaj *et al.*, "DataHub: Collaborative Data Science & Dataset Version Management at Scale," in *CIDR*, 2015.

[9] P. Buneman *et al.*, "Why and where: A characterization of data provenance," in *ICDT*, London, UK, Jan. 4–6, 2001, pp. 316–330.

[10] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," *The VLDB Journal*, vol. 12, no. 1, pp. 41–58, 2003.

[11] B. S. Arab *et al.*, "GProM—a swiss army knife for your provenance needs," *IEEE Data Eng. Bull.*, vol. 41, no. 1, 2018.

[12] D. Hernández *et al.*, "Computing how-provenance for SPARQL queries via query rewriting," *PVLDB*, vol. 14, no. 13, pp. 3389–3401, 2021.

[13] M. H. Namaki *et al.*, "Vamsa: Automated provenance tracking in data science scripts," in *Proc. KDD*, 2020.