Leveraging machine learning techniques for discovering broken lineage links between database objects

Paweł Boiński
Poznan University of Technology
Poznań, Poland *pawel.boinski@cs.put.poznan.pl*

Witold Andrzejewski
Poznan University of Technology
Poznań, Poland *witold.andrzejewski@cs.put.poznan.pl*

Miłosz Grocholewski
Poznan University of Technology
Poznań, Poland *milosz.grocholewski@gmail.com*

Tobiasz Gruszczyński
Poznan University of Technology
Poznań, Poland *t_gruszczynski@op.pl*

Robert Wrembel
Poznan University of Technology
Poznań, Poland *robert.wrembel@cs.put.poznan.pl*

Abstract

Data lineage is the set of techniques for tracking the flow of data throughout its lifecycle. These techniques are crucial for data management, governance, and compliance with regulations. Lineage links are maintained between data and database objects, but they are often broken by temporary objects and user defined functions. To the best of our knowledge, discovering broken lineage links has not been addressed yet in research. In this paper, we present a method for detecting broken lineage links between database objects. To this end we apply machine learning techniques on available metadata. We extract feature vectors and employ a classification approach to determine whether one database object is a source for another. Initial experiments on large database schemas show that the discovery of broken lineage links is possible at an acceptably high probability.

Keywords: data lineage, classification, metadata, broken lineage link.

1. Introduction and motivation

Digital transformation involves transforming company's internal processes to make use of data engineering, data management, and data analytics technologies, with the final goal to increase company's efficiency and improve customer relations. It relies on the ability of a company to build a complex information system, which typically integrates multiple heterogeneous and distributed data sources. Data integration technologies have been researched for decades (see for example [4, 5], [8], [13], [16], [18]). Key challenging factors in data integration include: (1) methods for homogenizing heterogeneous data, (2) methods for discovering erroneous data and correcting them (data cleaning), (3) providing efficient access to integrated data, and (4) providing means for tracing data processing from their source to destination (a.k.a. data lineage, data provenance). Two important aspects of data lineage are crucial for successful digital transformation: (1) trust and reliability as well as (2) compliance with regulatory demands.

Trust and reliability are essential in implementing the properly functioning automation of business processes. Data-driven decision making, inherent to such an approach, is only good as the data supplied. Lack of the data lineage information inhibits the verification of its correctness and applicability to the specific process. Moreover, decisions based on low-quality data are risky and highly uncertain. Failure of such automated business processes can cause adverse effects such as loss of income, client dissatisfaction, company's loss of credibility and, in extreme cases, compliance issues.

Compliance issues can be caused by company's failure to abide by regulatory demands. Data processing must adhere to multiple strict regulations (e.g., GDPR, CCPA, HIPAA). These require companies to report on how the data was collected, processed and used. Thus, audit trails provided via data lineage are necessary to meet these obligations. Providing the full lineage record for financial reports is an obligation in the financial sector, which is strictly regulated by means of European law, national law, and recommendations issued by institutions controlling the sector.

Data lineage represents the set of techniques for tracking and (typically) visualizing the flow of data throughout its lifecycle, i.e., from a data source (typically OLTP systems) to its final destination (typically a data warehouse, data lake, or data lakehouse) [1], [11], [29]. These techniques are crucial for data management, governance, and compliance, since they allow to analyze data pipelines, show how data were processed by every single step of a data pipeline, asses data quality, and prove data correctness for various supervisory bodies (especially in the financial industry).

Data lineage techniques have been researched for decades and resulted in numerous solutions. The lineage information is typically represented either by: (1) annotations attached to source tuples or their individual attributes, e.g., [6, 7], [10], [12] or by (2) additional structures that relate source and final (after processing) data, e.g., [14], [22], [25]. Commercial implementations allow to track, analyze (the so-called impact analysis), and visualize lineage that is based mainly on primary-foreign key relationships between database objects, see [17], [21] for overviews. Data lineage techniques allow to record relationships not only between data but also database objects.

In real applications, data pipelines often produce auxiliary temporary objects (e.g., temporary tables), typically in a data stage, in a data warehouse architecture. Moreover, data pipelines for big data frequently use user defined functions (UDFs) that allow to implement code snippets in various programming languages. UDFs allow to implement a logic tailored to a specific, usually not typical, data processing problem, e.g., [9], [23]. UDFs also produce temporary table-like data (e.g., table functions). Temporary objects and data produced by UDFs cease to exist after a given pipeline ends its execution. As a consequence, **data lineage links get broken**. To the best of our knowledge, either the existing research solutions or commercial or open-source systems **do not support discovering broken lineage** links.

This research work was motivated by a real problem faced by the financial industry. They aim at building a solution for discovering lineage links between data warehouse objects. The scope of objects designated for analysis is considerable, comprising over 11,000 stored procedures/ functions, exceeding 2 million lines of code, more than 5,000 views/ materialized views, and tens of thousands of tables.

In this paper, we propose a **method for discovering broken lineage links** between database objects (Section 2). It is based on classification models learned from large database schemas. The performance of the method (in terms of F1) was assessed experimentally (Section 3). The results show that it is possible to discover broken lineage links with high probabilities. Consequently, new research path in data lineage were identified. They are outlined in Section 4.

2. Contribution: machine learning for discovering broken lineage

The solution presented in this paper is the continuation of a previous work presented in [15]. The previous work introduces an approach to solve the broken lineage problem by means of a binary classification algorithm (Random Forest). The model is learned based on several features describing the similarity of two database objects (i.e., tables and views).

In this paper, to improve lineage discovery, we use the same method but extend it with: (1) a new set of features for classification and (2) additional classification models beyond Random Forest. We also perform a wider range of experiments.

Originally, the similarity between two tables was quantified using three features (with Jaro-Winkler [28] as the string similarity measure): (1) similarity of object names, (2) average similarity of attribute name pairs between the two objects: a source and a target (excluding attribute names containing 'id'), and (3) the number of attribute names in the source object that exhibit a similarity greater than 0.9 with the target object's name.

In this paper, we propose the following list of features for each pair of source and target database objects (similarly as before, the string similarity measure used is Jaro-Winkler):

- similarity between the source and target object names,
- maximum similarity obtained when comparing the source object name with each of the target object's attribute names,
- maximum similarity obtained when comparing the target object name with each of the source object's attribute names,
- maximum, minimum, mean, and median similarity between target object attribute names and source object attribute names.

The training and testing datasets used in this study are generated by analyzing the database creation scripts. Such scripts contain commands that create tables, views, or temporary tables using SELECT statements. The specific types of commands include:

- CREATE TABLE,
- CREATE TABLE ... AS SELECT,
- CREATE TEMPORARY TABLE ... AS SELECT,
- CREATE VIEW

If a database object is a table created by command CREATE (TEMPORARY) TABLE ... AS SELECT, or it is a view, then source object names used in the defining commands are extracted. Next, a graph is built, where vertices represent database objects, while arcs represent connections between source and target objects.

For each vertex representing a target database object (the vertex is incident with at least one incoming arc), all its source objects are retrieved. If the source object is a temporary table, then it is replaced with its source objects recursively, until only permanent source objects remain. This last step allows to find links (i.e., dependencies) from the target table to the permanent source database objects, which would normally be lost due to removal of intermediate temporary tables (broken lineage).

For example, see Fig. 1 and Fig. 2. The first figure presents a fragment of a database of a medical application. Actual dependencies among tables are represented as solid arrows. Consider a temporary table "SCHEDULED EXAMINATIONS" in Fig. 1 that assigns examinations to particular patients. The lineage of the data in the table includes records from tables "PATIENTS" and "EXAMINATIONS". During application runtime, the scheduled examinations

are moved to "COMPLETED EXAMINATIONS" table. Thus, the lineage of data in this table includes records from "SCHEDULED EXAMINATIONS" and, recursively, from "PATIENTS" and "EXAMINATIONS". However, since the table "SCHEDULED EXAMINATIONS" is temporary, its eventual removal creates a broken lineage problem.

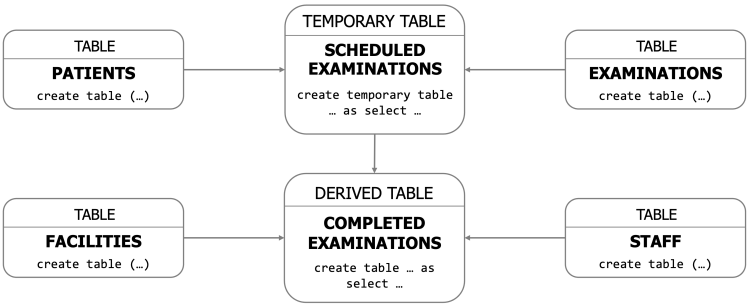

Fig. 1. Actual dependencies among tables

Fig. 2 presents such a situation. The "COMPLETED EXAMINATIONS" data lineage depends on still existing "PATIENTS" and "EXAMINATIONS" (dashed arrows), but this information is lost. The remaining dependencies are denoted by solid arrows.

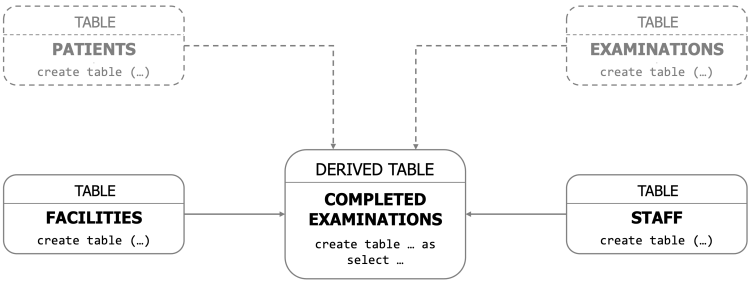

Fig. 2. Missing dependencies after removing temporary table "SCHEDULED EXAMINATIONS"

The broken lineage dependencies are simulated using the recursive mechanism previously described for generating dataset object pairs. For all the obtained source-target pairs the aforementioned similarity features are computed and are labeled with the decision attribute value 1 (later denoted as the *positive class*). For all (ordered) pairs of two permanent objects, such that they were not labeled previously, the feature values are computed as well and are labeled with the decision attribute value 0 (later denoted as the *negative class*).

The main problem with this approach, is that the dataset can be highly unbalanced, meaning there are many more negative class instances (0) than positive class instances (1). In our experiments the ratio of positive to negative classes was around 1:300. Thus, a suitable approach must be employed to handle the class imbalance effectively.

3. Experiments

The experiments were designed to evaluate several classifiers in addressing the broken lineage problem. We generated training and testing database schemas using Large Language Models (see Section 3.1). Next, for each pair of attributes (in both training and testing datasets), a set of features was generated, as described in Section 2. Finally, multiple different classification models for several different parameter sets, were trained using the training dataset. The trained models were applied to the testing dataset and quality metrics were computed. The three best models were chosen. We present their short description in Section 3.2 and the obtained results

in Section 3.3.

3.1. Training and testing database schemas

For the purpose of the experiments we have prepared database creation scripts via two different Large Language Models (LLMs), to model two distinct "database designers". The first database, representing a financial system, originally used for training in [15], was generated by Llama3 [2] model in the Ollama3 tool. The database was composed of 223 regular tables, 93 temporary tables, 93 materialized query tables and 53 views. In our experiments, we used it for testing.

To construct the second (training) database, several of the most popular free LLMs were evaluated. Ultimately, *Grok* [3] was selected due to its superior performance. Even though *Grok* turned out to produce the schema of the highest quality, the task of generating the schema proved to be challenging. The database schema was intended to include interdependent objects (related to other objects). Therefore, beyond standard tables, it was necessary to generate temporary tables and materialized query results (using CREATE TABLE ... AS SELECT commands).

We assumed that the target schema, representing the health management system database, would include 250 regular tables, 200 temporary tables, 200 materialized query result tables and 100 views. However, despite the advanced capabilities of the tested LLMs, including *Grok*, it was not possible to generate such a large and interconnected schema without employing iterative processing, even for the initial 250 standard tables (i.e., without dependencies based on ... AS SELECT commands). Definitions of subsequent tables that referenced existing database objects were frequently generated incorrectly, which required repeated human intervention and error correction. Moreover, the frequency of errors increased with the growing complexity of the schema. For example, among the final set of 100 materialized query result tables, approximately 50% were initially generated with significant inaccuracies.

SQL scripts for generated databases can be downloaded from `https://github.com/witold-andrzejewski/broken-provenance`.

The testing and training datasets were created as described in Section 2. The *training dataset* included: $1\,013$ positive rows representing existing dependencies between source and target database objects, and $299\,924$ negative rows representing pairs where the source object was not used to generate the target object. The *testing dataset* initially contained 408 positive rows and $134\,976$ negative rows. However, in order to balance the classes in the testing dataset, we randomly sampled only 408 negative rows to ensure the equal number of positive and negative instances.

3.2. Tested classifiers

We tested various classification models available in Python's *scikit-learn* and compatible packages. The best results were obtained from methods specifically designed to handle highly unbalanced datasets. The three best methods were as follows: *BalancedBaggingClassifier* and *RUSBoostClassifier* [24] (from package *imblearn* [20]) as well as *LGBMClassifier* [19] (from package *lightgbm* [26]).

BalancedBaggingClassifier is an implementation of the *bagging* approach with additional balancing of the training dataset by applying some sampling strategy. In our experiments, we used the default settings, in which the base estimator was a decision tree, the sampling strategy was based on random undersampling, and the training process encompassed bootstrapping.

RUSBoostClassifier is a modification of *AdaBoost* in which a random undersampling is performed at every iteration of the boosting algorithm. During the experiments, the best results were obtained when the number of trained estimators was increased to 100 and the base estimator was *Support Vector Machine*. Other parameters were set to default values.

LGBMClassifier is a modification of gradient boosting decision tree which includes sam-

Table 1. The performance of the tested classifiers

RUSBoostClassifier		Accuracy	0.78	Confusion matrix		
Class	Precision	Recall	F1-Score		Predicted 0	Predicted 1
0	0.75	0.83	0.79	True 0	337	71
1	0.81	0.73	0.76	True 1	112	296
LGBMClassifier		Accuracy	0.72	Confusion matrix		
Class	Precision	Recall	F1-Score		Predicted 0	Predicted 1
0	0.66	0.92	0.77	True 0	375	33
1	0.87	0.53	0.66	True 1	192	216
BalancedBaggingClassifier		Accuracy	0.703	Confusion matrix		
Class	Precision	Recall	F1-Score		Predicted 0	Predicted 1
0	0.71	0.69	0.70	True 0	281	127
1	0.70	0.72	0.71	True 1	115	293

pling of data instances based on gradients and special treatment of exclusive features. In our experiments, the best results were obtained for the default settings, with the exception of the number of estimators reduced to 10 and the learning rate reduced to 0.05. Moreover, class weights were changed to 1 for the negative and to 100 for the positive class, to counter the class imbalance problem.

3.3. Results

The results of experimental evaluation are presented in Table 1. The best results were obtained by *RUSBoostClassifier*, with accuracy up to 0.78 and F1-scores better than the two other models. Moreover, one can easily notice that the results of *BalancedBaggingClassifier* are completely dominated by *RUSBoostClassifier* (every numerical result is better). Nonetheless, both of these classifiers have the advantage that roughly the same numbers of database object pairs were predicted to be either the positive or the negative class. For *RUSBoostClassifier*, 449 pairs of database objects were predicted as negative and 367 were predicted as positive, while for *BalancedBaggingClassifier*, 396 pairs were predicted as negative and 420 as positive.

Since the testing dataset was also balanced, this led to balanced precision and recall values. *LGBMClassifier* achieved accuracy of 0.72, which is in-between the two previously described classifiers. Unfortunately, the results for this classifier are more unbalanced, as 567 pairs were predicted as negative and 249 as positive. This shows the tendency of the trained model to predict the majority (negative) class and causes the recall of negative class and precision of the positive class to be increased at the cost of precision of the negative class and recall of the positive class.

4. Summary and future work

In this paper, we present a classification-based approach to metadata-based broken lineage detection. In particular, we evaluate a feature list for comparing the metadata of two database objects (permanent tables, temporary tables, and materialized query results). These features, when combined with the training lineage data, are used to train classifier models. Our solution is evaluated on many different classifiers. The obtained results show that it is possible to predict broken lineage links with the accuracy up to 0.78 and F1 up to 0.79 (*RUSBoostClassifier*).

Notice that the experiments were run on large, synthetically generated OLTP-like schema. Since our method proved to be satisfactory, in the next step, we plan on training and testing models on historical lineage links acquired from code repositories (e.g., GitHub, GitLab), recognized database benchmarks (e.g., tpc.org) and from our own databases. The work presented here opens a new research path in data lineage.

The first task is to verify how our method performs under dirty metadata describing database schemas. Second, we plan on testing a new, LLM-based approach for comparing the attribute

and database object names. In particular, we plan of normalizing each such name by extracting and extending each compound word or acronym into full size. For example, "emp_main_sal" attribute name, can be converted into a list: ["employee", "main", "salary"]. Each such word can be additionally enriched with a list of its synonyms. Similarity between names can be computed based on how many words agree on at least one synonym and how many words do not. Third, as an alternative, we envisage comparing the embeddings [27] of compound words to determine whether the embeddings of words used in two names are similar. Fourth, we plan on increasing heterogeneity of training and testing database schemas via prompt engineering or using multiple distinct LLMs similarly as in Section 3.1 as well as including real-world datasets. Next, we will explore the possibility to extend the classification model with new features like schema patterns (e.g., OLTP, OLAP) and value patterns. We will also discover misclassification patterns and test how the degree of class imbalance impacts the results. Finally, we plan to extend the set of classification evaluation metrics to accurately analyze the quality of generated models.

References

[1] What is data lineage?, IBM documentation; `https://www.ibm.com/topics/data-lineage`

[2] Llama 3: Open foundation and instruction models. `https://ai.meta.com/llama/` (2024)

[3] Grok xAI, version 3 (2025), `https://x.ai/grok`

[4] Andrzejewski, W., Bebel, B., Boinski, P., Wrembel, R.: On tuning parameters guiding similarity computations in a data deduplication pipeline for customers records: Experience from a r&d project. Information Systems 121 (2024)

[5] Azzini, A., Jr., S.B., Bellandi, V., Catarci, T., Ceravolo, P., Cudré-Mauroux, P., Maghool, S., Pokorny, J., Scannapieco, M., Sedes, F., Tavares, G.M., Wrembel, R.: Advances in data management in the big data era. IFIP AICT, vol. 600, pp. 99–126. Springer (2021)

[6] Benjelloun, O., Sarma, A.D., Halevy, A.Y., Theobald, M., Widom, J.: Databases with uncertainty and lineage. VLDB Journal 17(2), pp. 243–264 (2008)

[7] Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An annotation management system for relational databases. VLDB Journal 14(4), pp. 373–396 (2005)

[8] Birgersson, M., Hansson, G., Franke, U.: Data integration using machine learning. In: IEEE Int. Enterprise Distributed Object Computing Workshop (EDOC). pp. 1–10 (2016)

[9] Chasialis, K., Palaiologou, T., Foufoulas, Y., Simitsis, A., Ioannidis, Y.E.: Qfusor: A UDF optimizer plugin for SQL databases. In: ICDE. pp. 5457–5460 (2024)

[10] Chiticariu, L., Tan, W.C., Vijayvargiya, G.: DBNotes: a post-it system for relational databases based on provenance. In: SIGMOD. p. 942–944 (2005)

[11] Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. VLDB Journal 12(1), pp. 41–58 (2003)

[12] Dosso, D., Davidson, S.B., Silvello, G.: Data provenance for attributes: attribute lineage. In: USENIX Conf. on Theory and Practice of Provenance. USENIX Assoc. (2020)

[13] Errami, S.A., Hajji, H., Kadi, K.A.E., Badir, H.: Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse. Journal of Parallel and Distributed Computing 176, pp. 70–79 (2023)

[14] Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS. p. 31–40 (2007)

[15] Grocholewski, M., Gruszczynski, T.: Structure lineage for database objects and code: design, implementation, and experimental evaluation. Master thesis (in polish), Poznan University of Technology (2024)

[16] Hai, R., Koutras, C., Quix, C., Jarke, M.: Data lakes: A survey of functions and systems (extended abstract). In: Int. Conf. on Data Engineering (ICDE). pp. 5679–5680 (2024)

[17] Hariharan, A., Zhang, T., Motz, M., Weinhardt, C.: Accessible data lineage: A scoping review on open-source data lineage platforms. In: ICIS (2024)

[18] Ilyas, I.F., Rekatsinas, T.: Machine learning and data cleaning: Which serves the other? ACM Journal of Data and Information Quality 14(3) (2022)

[19] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: LightGBM: A highly efficient gradient boosting decision tree. In: Advances in Neural Information Processing Systems. vol. 30, p. 3149–3157. Curran Associates, Inc. (2017)

[20] Lemaître, G., Nogueira, F., Aridas, C.K.: Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. Journal of Machine Learning Research 18(17), pp. 559–563 (2017)

[21] Ponnusamy, S., Gupta, P.: Connecting the dots: How data lineage helps in effective data governance. Int. Journal on Computer Science and Engineering 10(10), pp. 6–10 (2023)

[22] Psallidas, F., Wu, E.: Smoke: Fine-grained lineage at interactive speed. VLDB Endowment 11(6), pp. 719–732 (2018)

[23] Schäfer, N., Gjurovski, D., Davitkova, A., Michel, S.: To UDFs and beyond: Demonstration of a fully decomposed data processor for general data wrangling tasks. VLDB Endowment 16(12), pp. 4018–4021 (2023)

[24] Seiffert, C., Khoshgoftaar, T.M., Van Hulse, J., Napolitano, A.: Rusboost: A hybrid approach to alleviating class imbalance. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 40(1), pp. 185–197 (2010)

[25] Senellart, P.: Provenance and Probabilities in Relational Databases. SIGMOD Record 46(4), pp. 5–15 (2018)

[26] Shi, Y., Ke, G., Soukhavong, D., Lamb, J., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y., Titov, N., Cortes, D.: LightGBM: Light Gradient Boosting Machine (2025), https://github.com/Microsoft/LightGBM

[27] Wagan, A.A., Li, S.: Multilabeled emotions classification in software engineering text using convolutional neural networks and word embeddings. J. of Software: Evolution and Process 37(3) (2025)

[28] Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In: Proceedings of the Survey Research Methods Sections. pp. 354–359. American Statistical Association (1990)

[29] Yamada, M., Kitagawa, H., Amagasa, T., Matono, A.: Augmented lineage: traceability of data analysis including complex UDF processing. The VLDB Journal 32(5), pp. 963–983 (2023)