The VLDB Journal (2023) 32:963–983
https://doi.org/10.1007/s00778-022-00769-7

REGULAR PAPER

Augmented lineage: traceability of data analysis including complex UDF processing

Masaya Yamada[1,3] · Hiroyuki Kitagawa[2,3] · Toshiyuki Amagasa[4] · Akiyoshi Matono[3]

Received: 13 March 2022 / Revised: 4 September 2022 / Accepted: 24 October 2022 / Published online: 23 November 2022

Abstract

Data lineage allows information to be traced to its origin in data analysis by showing how the results were derived. Although many methods have been proposed to identify the source data from which the analysis results are derived, analysis is becoming increasingly complex both with regard to the target (e.g., images, videos, and texts) and technology (e.g., AI and machine learning (ML)). In such complex data analysis, simply showing the source data may not ensure traceability. For example, ML analysts building image classifier models often need to know which parts of images are relevant to the output and why the classifier made a decision. Recent studies have intensively investigated interpretability and explainability in the AI/ML domain. Integrating these techniques into the lineage framework will help analysts understand more precisely how the analysis results were derived and how the results are trustful. In this paper, we propose the concept of *augmented lineage* for this purpose, which is an extended lineage, and an efficient method to derive the augmented lineage for complex data analysis. We express complex data analysis flows using relational operators by combining user-defined functions (UDFs). UDFs can represent invocations of AI/ML models within the data analysis. Then, we present a method taking UDFs into consideration to derive the augmented lineage for arbitrarily chosen tuples among the analysis results. We also experimentally demonstrate the efficiency of the proposed method.

Keywords Traceability · Lineage · User-defined function · Complex data analysis · Augmented lineage

1 Introduction

When data analysis is utilized for decision-making, it is critical to guarantee the traceability of the results. Data provenance in data analysis refers to all metadata describing its processing and is essential for traceability. It has been discussed in many research domains such as debugging [30,37], security [12], scientific workflow [24], and visualization [10]. In particular, data lineage refers to tracing the source data from which the analysis results are derived. This topic has been widely researched in the database domain [16,17,53].

Even with lineage, traceability remains a challenge. The assumption for the conventional lineage is that users can understand why the output was derived only by observing the source data. This assumption holds if the analysis consists of a sequence of simple operators (e.g., filter, join, and aggregation) and their operational semantics are simple. However, modern data analysis has become more complex. Not only are the targets composed of diverse content data such as images, videos, and texts but processing often involves sophisticated technologies such as AI and machine learning (ML). In such

✉ Masaya Yamada
yamada@kde.cs.tsukuba.ac.jp

Hiroyuki Kitagawa
kitagawa@cs.tsukuba.ac.jp

Toshiyuki Amagasa
amagasa@cs.tsukuba.ac.jp

Akiyoshi Matono
a.matono@aist.go.jp

1 Graduate School of Science and Technology, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

2 International Institute for Integrative Sleep Medicine, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

3 Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology, 2-4-7 Aomi, Koto-ku, Tokyo, Japan

4 Center for Computational Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

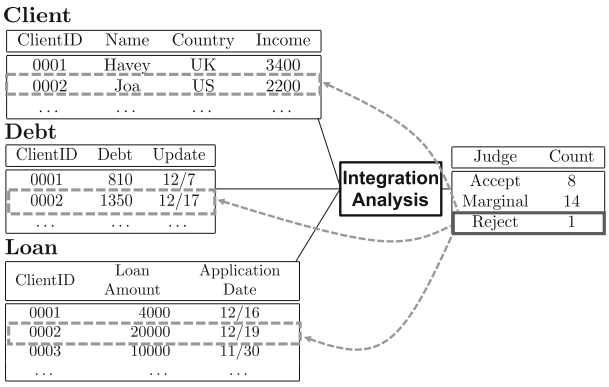


Fig. 1 Analysis summarizing the results of examinations for loan application applied on December using an ML model. Tuples surrounded by dashed lines represent the source data of the result tuple ⟨Reject, 1⟩

complex analysis, users cannot understand the reason for the analysis result only from the lineage (source data). In other words, we need more information about the reasoning basis for AI/ML processing. Consider the following example.

Example 1 Figure 1 shows an analysis of a financial institution summarizing the results of examinations for loan application applied on December based on customer information using an ML model. The ML model takes three attribute values (Income, Debt, LoanAmount) as inputs and returns the examination result (Accept, Marginal, Reject). The reason that an application was rejected is not obvious only from the input. In this case, the lineage of ⟨Reject, 1⟩ is a set of tuples {⟨0002, Joa, US, 2200⟩, ⟨0002, 1350, 12/17⟩, ⟨0002, 20000, 12/19⟩}. This is insufficient to understand why ⟨Reject, 1⟩ was derived from these source tuples. In other words, the source tuples do not fully explain "why this application was rejected." However, the reason for the decision in the ML model along with the source tuples may explain why an application was rejected. For example, "the model rejected this application because the debt and loan amount of the applicant are both large." This reason describes the basis of the result, and it can help people understand the model behavior leading to traceability for complex data analysis.

In this paper, we propose the concept of *augmented lineage*, which is an extension of lineage that incorporates the basis for computation (*reason*) in complex data analysis. Reason provides information which helps users understand the computation, for example, (1) which region of the content data (e.g., images or videos) the analytical model emphasizes and (2) which attributes much affect the AI/ML model decision. Since reason is essential in our augmentation of lineage, we use the term "augmented lineage" in the sense of "reason-augmented lineage" in this paper. The contributions of this paper are as follows: (i) proposal and formulation of augmented lineage, (ii) proposal of a basic algorithm to derive the augmented lineage, (iii) proposal of an enhanced method for its efficient derivation, and (iv) experimental evaluation of the proposed derivation method.

The preliminary version of this paper was published in [55]. We have intensively extended our previous work in the following three points: (i) This paper includes detailed descriptions of our model, augmented lineage, and algorithms to derive augmented lineage. (ii) We fully revised our segmentation scheme for augmented lineage derivation. The revised one preserves the order of operators in a given operator tree as it is. This feature is beneficial if the given operator tree is optimized by the query optimizer. (iii) This paper includes extensive experiments based on the derived TPC-H benchmark databases and queries of three scale factors [50] as well as the original experiments based on LFW image dataset [28].

The rest of this paper is organized as follows: Sect. 2 overviews related work. Section 3 presents the data model to represent data analysis in this paper. Section 4 proposes the augmented lineage, while Sect. 5 formulizes a basic algorithm to derive the augmented lineage. Section 6 details its enhanced method and the system architecture. Section 7 experimentally investigates the derivation method. Finally, Sect. 8 concludes this paper.

2 Related work

Provenance and lineage have been researched widely in many domains [8,13,27].

[4–6,14,15,17,23,43,52] proposed methods that derive source data for database queries. [52] derived lineage of output values using user-defined weak inversion functions. In the framework, the derived lineage depends on the inversion functions the user provides and the framework provides, and it may derive incomplete and/or redundant lineage. [9] discussed two granularities for provenance: tuple-level (Why-Provenance) and value-level (Where-Provenance). Value-level lineage was derived in [52], and tuple-level lineage was derived in [15,17]. [15,17] proposed a method to derive accurate tuple-level lineage for relational queries by performing tracing queries. These frameworks focused on deriving the lineage of specified output tuples after query execution. Such a framework is called the *lazy* approach. The lazy approach has the feature that it takes no/small overhead for the ordinary query execution, but it takes some extra work to derive lineage after the query.

There is an alternative approach called the *eager* approach. Frameworks based on the eager approach generally derive provenance/lineage for all output tuples while executing the analysis. Therefore, it incurs some overhead for the ordinary analysis execution, but the provenance/lineage is ready as well as query results. [4–6,14,23,43] are frameworks based

on the eager approach. Trio [4,5] attached identifiers to all source tuples and inherited them as annotations to query results during the analysis to derive tuple-level lineage. They also considered the application of the lineage for the uncertain data model and probabilistic data model. DBNotes [6,14] attached value IDs to all attribute values in source tables, inherited the IDs during the query execution, and finally captured value-level provenance. Perm [23] derived tuple-level lineage by rewriting original relational queries so that they keep information about attributes in the source tuples as well. Smoke [43] instrumented relational operators to construct lineage indexes, which maintain the relationship between input and output tuples while executing the query. Using the indexes, it could identify the input tuples from which output tuples were derived (backward lineage) and the output tuples to which input tuples contributed (forward lineage).

[3,25,47] discussed more sophisticated annotation-based provenance models known as the semiring provenance model. [25] firstly proposed the model. Annotations were attached to the source tuple and inherited during analysis, and polynomials on semiring were derived as provenance. Following this, [3] extended the framework to handle aggregate operation. When the database size is huge or the complex query is executed, the polynomial for semiring provenance becomes quite huge, leading to difficulty in understanding provenance information. Some frameworks addressed the problem. [20] represented semiring-based provenance using the circuit, which reduced the provenance size, and [1] focused on approximating the provenance with practical information loss. In addition to the above frameworks, some studied provenance in other types of database queries (e.g., Datalog [19,25], XQuery [22], and SPARQL [49]). Traceability has been researched in domains other than the database.

In the scientific workflow, [7,24] captured the provenance for reproducibility of the computation. Recently, some researches targeted the distributed big data processing systems and proposed how to capture provenance (e.g., MapReduce [2,36], Spark [30] and Flink [41,42]). In addition, provenance has been also researched in the ML domain, and [39,48,54] focused on the provenance for the training phase of ML models.

Modern data analysis includes content data analysis and AI/ML analysis (e.g., recommendation [56], anomaly detection [11,33], and medical diagnosis [32,35]). There are some researches that can handle traceability in such complex data analysis. [16,53] derived the lineage for the analysis which included user-defined functions (UDFs) and relational operators. [16] proposed a method to derive the lineage for analysis which included more generalized operators. Their method can deal with non-relational operators by focusing on the relationship between the input and output data (e.g., one-to-one, N-to-one, N-to-M relationships) of each operator and schema information. [53] presented a method to find lineage in SciDB, which handles multi-dimensional array data for analysis involving user-defined operators. Their study could identify backward lineage and forward lineage at the cell level. The lineage was obtained by recording the input/output relationships of each operator in the workflow. They proposed a method to efficiently manage and later derive lineages by rerunning. In these studies, lineage means the correspondence between input/output tuples (cells). However, they did not show the important parts of the content data (images, texts, etc.). Moreover, they did not cover the reasoning basis for AI/ML decisions. [57] proposed a method to track information about which parts of the content data were extracted by data extraction processing represented by UDFs as well as the traditional tuple-level provenance based on the semiring model. Even this framework cannot provide the basis for decisions made by AI/ML processing.

Recently, many studies have investigated interpretability and explainability of AI/ML processing [21,26]. Using frameworks such as [38,45,46], the basis for the decisions in AI/ML processing can be shown. The augmented lineage presented in this paper is a general framework that integrates such reasoning functions into lineage and contributes to the traceability suitable for complex data analysis, including AI/ML processing.

Our framework lazily derives augmented lineage, and it uses [17] as a base reference model. [17] is a state-of-the-art approach to lazily derive tuple-level lineage for relational queries. However, our framework extends the base model in the following points: (1) Inclusion of an additional operator (Function operator) to model complex data analysis. (2) Augmentation of lineage by reason to show the basis of complex data analysis. (3) Proposal of augmented lineage derivation methods. (4) Intensive experimental evaluation of augmented lineage derivation performance.

3 Data model

This section describes the data model to represent data analysis. Complex data analysis is modeled as a *task* consisting of relation-like operators.

3.1 Data

We model data as a collection of tables. Table $T(A_1, \ldots, A_n)$ has a set of tuples $\{t_1, \ldots, t_p\}$. $\boldsymbol{A}_T$ denotes all attributes of a table T. $t.\boldsymbol{A}$ projects tuple t onto the tuple with an attribute set $\boldsymbol{A}$ ($\subseteq \boldsymbol{A}_T$). For a set of tuples $\{t_1, \ldots, t_q\}$ (including, in special cases, single value instead of t_i), $\langle t_1, \ldots, t_q \rangle$ denotes the tuple that concatenates them. For example, for $t_1 = \langle 1, 2 \rangle, t_2 = \langle 3 \rangle, \langle t_1, t_2 \rangle = \langle 1, 2, 3 \rangle$. Source data set

D consists of tables $(T_1, \ldots, T_m)$.[1] $\mathcal{T}$ denotes a task which consists of operators defined in Sect. 3.2 on the source data set D, and O denotes its output. That is, $O = \mathcal{T}(D) = \mathcal{T}(T_1, \ldots, T_m)$. A task can consist of no operator as a special case. In this case, the output is a source table itself, $O = T_i$.

3.2 Operators

We describe the operators that compose a task. To model data analysis, we assume seven set-relational operators: basic operators (Selection σ, Projection π, Join $\bowtie$, Aggregation α, Union $\cup$, Difference $-$) and Function operator ϕ, which models complex data analysis. A task is represented as a tree consisting of these operators, while leaf nodes are source tables. First, we define each basic operators as follows:

Definition 1 (*Basic operators*)

- Selection: $\sigma_C(T) = \{t \mid t \in T \wedge t \text{ satisfies condition } C\}$.
- Projection: $\pi_{\boldsymbol{A}}(T) = \{t.\vec{A} \mid t \in T\}$. Note that its results are duplicate eliminated.
- Join: $T_1 \bowtie_{\theta_1} \cdots \bowtie_{\theta_{m-1}} T_m = \{\langle t_1, \ldots, t_m\rangle \mid t_i \in T_i\ (i = 1, \ldots, m) \wedge \langle t_1, \ldots, t_m\rangle \text{satisfies condition}\, \theta_i (i = 1, \ldots, m-1)\}$. Join operator without θ_i represents natural join.
- Aggregation: $\alpha_{\boldsymbol{G}, g(\boldsymbol{B})}(T) = \{\langle T'.\boldsymbol{G}, g(T'.\boldsymbol{B})\rangle \mid T' \subseteq T, \forall t, t' \in T', \forall t'' \notin T' : t'.\boldsymbol{G} = t.\boldsymbol{G} \wedge t''.\boldsymbol{G} \neq t.\boldsymbol{G}\}$. Note that $\boldsymbol{G}.$ denotes the grouping key of T and $g(\boldsymbol{B})$ denotes an aggregate function over attribute $\boldsymbol{B}$ in the grouped table.
- Union: $T_1 \cup \cdots \cup T_m = \{t \mid t \in T_i\ (i = 1, \ldots, m)\}$.
- Difference: $T_1 - T_2 = \{t \mid t \in T_1 \wedge t \notin T_2\}$.

To model complex data analysis, we will employ UDF and the function operator to invoke UDF. Before explaining them, we define *reason*.

Definition 2 (*Reason*) Reason is a data object returned by each UDF invocation to help users understand the UDF computation. UDF developers are responsible for making each invocation of UDF output both a computation result and a reason.

Our definition of reason here is very general, and any data object which may help users understand the behavior of the computation and model can be defined as a reason. In the domain of explainable AI models, various types of data are used as "explanations" or "reasons" depending on models and problems [21,31]. To make our model applicable to different domains, we employ this rather general definition here. Recently, model-agonistic explanation models such as LIME [45] and SHAP [38] are attracting attention. In those approaches, interpretable approximations of the original model are used as explanation models, and weights associated with simplified features of the explanation model are often deployed as "explanations". Our definition can accommodate such "explanations" when the UDF developer thinks they are useful to help understand the UDF computation.

Example 2 (Reason) Let us consider a classifier that takes customer and loan application information as input to examine the application, like in Example 1. Suppose a developer wants the classifier to show which input attributes contribute a lot in decision-making as well as the decision results to help understand the basis for the decision. In that case, the developer can make a data object specifying significant input attributes as a reason.

Now, we define UDF that models complex data analysis and is invoked in a function operator in our model. UDF has two execution modes (normal mode and reasoning mode). Intuitively, given input attribute values, the normal mode only returns the result of complex data analysis. On the other hand, the reasoning mode returns reasons as well as the results.

Definition 3 (*UDF*) UDF is the function, which has the following input and output depending on its execution mode.

1. Normal mode

$$f_n : \text{Domain}(\boldsymbol{E}) \rightarrow \mathit{Value}$$

2. Reasoning mode

$$f_r : \text{Domain}(\boldsymbol{E}) \rightarrow \mathit{Value} \times \mathit{Reason}$$

Here, Domain($\boldsymbol{E}$) is the domain of attributes $\boldsymbol{E}$, *Value* is the domain of the output values of the complex data analysis, *Reason* represents the domain of reasons, and $\times$ denotes the cross product of each domain. In the *normal mode*, UDF invocation outputs just the calculation result of the UDF, while, in the *reasoning mode*, it produces the reason for the calculation result as well. The UDF developers have to design and determine what information is produced as reasons. In more detail, our framework assumes that the developer should implement two functions for a UDF: *main*(), which is called in the normal mode invocation and returns only the UDF computation result, and *main_with_reason*(), which is called in the reasoning mode invocation and returns a reason as well. We assume that UDFs invoke pre-trained AI/ML models and their training is out of the scope of this paper.

Depending on the property and implementation of UDF, providing reason information may need much extra computation and be time-consuming processing. In addition, as we

[1] If a table is referred to more than once in a task, each reference is regarded as access to a different table.

discuss later, it is not always needed for UDF to generate reasons even in deriving augmented lineage. The two execution modes of UDF provide such choices.

Example 3 (UDF) In Fig. 1, we employ *credit_exam* function as UDF, which takes attribute values of Income, Debt, and LoanAmount of a client, and then returns the examination result (Accept, Marginal, Reject). Besides, it outputs the set of attributes contributing a lot to the result as a reason when it is invoked in reasoning mode. For example, *credit_exam* function returns the following output(s) for the client with ClientID "0001":

1. Normal mode

$$credit\_exam_n(\{3400, 810, 4000\}) = \text{"Accept"}$$

2. Reasoning mode

$$credit\_exam_r(\{3400, 810, 4000\}) = [\text{"Accept"}, \{\text{Income, LoanAmount}\}]$$

Then, we define the function operator. Intuitively, function operator adds the result (Value) of the UDF as a new attribute value to the input tuple when invoked in the normal mode, and adds Value and Reason of the UDF as new attribute values when invoked in the reasoning mode.

Definition 4 (*Function operator*) Function operator $\phi_{f(\boldsymbol{E})}$ applies UDF f to all tuples (more precisely, to their values for attributes $\boldsymbol{E}$) in the input table. The output of function operator $\phi_{f(\boldsymbol{E})}$ is as follows depending on UDF's execution mode:

1. Normal mode

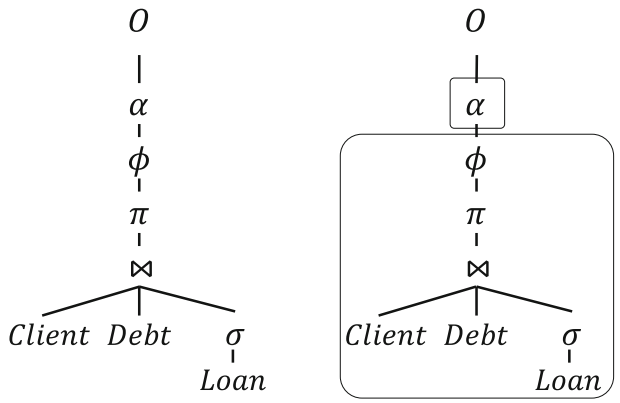
$$\begin{aligned}\phi^n_{f(\boldsymbol{E})}(T) &= \phi_{f_n(\boldsymbol{E})}(T)\\ &= \{\langle t,\ f_n(t.\boldsymbol{E}).\text{Value}\rangle | t \in T\}\end{aligned}$$

2. Reasoning mode

$$\begin{aligned}\phi^r_{f(\boldsymbol{E})}(T) &= \phi_{f_r(\boldsymbol{E})}(T)\\ &= \{\langle t,\ f_r(t.\boldsymbol{E}).\text{Value},\ f_r(t.\boldsymbol{E}).\text{Reason}\rangle | t \in T\}\end{aligned}$$

Example 4 (Analysis modeling and function operator) The analysis in Fig. 1 can be modeled as follows:

$$\begin{aligned}&O = \mathscr{T}(D) = \alpha_{\text{Judge,COUNT}(*)}\\ &(\phi_{f(\boldsymbol{E})}(\pi_{\boldsymbol{A}}(\text{Client} \bowtie \text{Debt} \bowtie \sigma_C(\text{Loan}))))\\ &\text{s.t. } \boldsymbol{A} = \boldsymbol{E} = \{\text{Income, Debt, LoanAmount}\}\\ &C : \text{Application date is on December}\end{aligned}$$



(a) Original operator tree **(b)** Segments of the task

Fig. 2 Segmentation of the operator tree

Figure 2a shows its operator tree, and Fig. 3 shows its intermediate results.

The function operator invokes *credit_exam* UDF in Example 3. Here, "Judge" corresponds to the UDF's Value, and * denotes all the attributes of the input table. In the reasoning mode, the function operator returns a table in Fig. 3e."Reason" corresponds to the UDF's Reason.

For ease of exposition, this paper restricts the operators to relational operators. However, our framework can deal with more generalized processing and external programs as long as they have the same input/output relationships as relational operators. For example, when an external program inputs each tuple and (1) outputs it or (2) discards it, then we can model this processing as selection σ because it has the same input/output relationship. Similarly, other data processing can be modeled as one of the above operators as long as their input/output relationship is the same.

4 Augmented lineage

This section defines *augmented lineage*. Augmented lineage is an extension of *lineage* in [15]. The set of the source tuples from which tuple $o(\in \mathscr{T}(D))$ of task $\mathscr{T}$ is derived is called the *source lineage* of tuple o in task $\mathscr{T}$.

We first define the source lineage of tuple o in an operator Op, and then define the source lineage of tuple o in task $\mathscr{T}$. The following definition specifies three features: (1) The tuples in the source lineage produce output tuple o. (2) Every tuple in the source lineage contributes to deriving tuple o. (3) The source lineage is the maximum subsets of source tables that meet conditions (1) and (2).

Definition 5 (*Source Lineage in an Operator*) Let $O(= Op(T_1, \ldots, T_m))$ be the result of performing an operator Op on the set of tables $\{T_1, \ldots, T_m\}$. Source lineage of tuple

ClientID	Name	Country	Income
0001	Havey	UK	3400
0002	Joa	US	2200
...	...	...	...

ClientID	Debt	Update
0001	810	12/7
0002	1350	12/17
...	...	...

ClientID	Loan Amount	Application Date
0001	4000	12/16
0002	20000	12/19
0003	10000	11/30
...	...	...

(a) Base tables (Left: Client table, Center: Debt table, Right: Loan table)

ClientID	Loan Amount	Application Date
0001	4000	12/16
0002	20000	12/19
...	...	...

(b) Result of $\sigma(Loan)$

ClientID	Name	Country	Income	Debt	Update	Loan Amount	Application Date
0001	Havey	UK	3400	810	12/7	4000	12/16
0002	Joa	US	2200	1350	12/17	20000	12/19
...	...	...	...	...	...	...	...

(c) Result of $Client \bowtie Debt \bowtie \sigma(Loan)$

Income	Debt	LoanAmount
3400	810	4000
2200	1350	20000
...	...	...

(d) Result of $\pi(Client \bowtie Debt \bowtie \sigma(Loan))$

Income	Debt	LoanAmount	Judge	Reason
3400	810	4000	Accept	{Income, LoanAmount}
2200	1350	20000	Reject	{Debt, LoanAmount}
...	...	...	...	...

(e) Result of $\phi(\pi(Client \bowtie Debt \bowtie \sigma(Loan)))$

Judge	Count
Accept	8
Marginal	14
Reject	1

(f) Final result of the task

Fig. 3 Base tables, intermediate results, and final result of the task

$o(\in O)$ in operator $Op(SL_{Op}(o))$ is the set of $\hat{T}_i \subseteq T_i$ $(i = 1, \ldots, m)$ which meets the following conditions.[2]

1. $Op(\hat{T}_1, \ldots, \hat{T}_m) = \{o\}$
2. $\forall \hat{T}_i : \forall \hat{t} \in \hat{T}_i : Op(\hat{T}_1, \ldots, \{\hat{t}\}, \ldots, \hat{T}_m) \neq \emptyset$
3. $\hat{T}_i$ is the maximum subset of T_i that meets 1 and 2.

The source lineage of tuple set $\bar{O}(\subseteq O)$ in operator Op is $SL_{Op}(\bar{O}) = \bigcup\!\!\!\!\bigcup_{o \in \bar{O}} SL_{Op}(o)$. Note that $\bigcup\!\!\!\!\bigcup$ produces the union for each table: $\{T_1^1, \ldots, T_m^1\} \bigcup\!\!\!\!\bigcup \{T_1^2, \ldots, T_m^2\} = \{T_1^1 \cup T_1^2, \ldots, T_m^1 \cup T_m^2\}$ s.t. $T_i^j \subseteq T_i$. $\hat{T}_i \in SL_{Op}(\bar{O})$ is denoted as $P_{T_i}(SL_{Op}(\bar{O}))$.

Example 5 (Source lineage in an operator) Let us consider the join operator $\bowtie$ in Fig. 2a, which produces the table shown in Fig. 3c. The source lineage of tuple $(\langle 0001, \ldots, 12/16\rangle)$ in the join is $SL_{\bowtie}(\langle 0001, \ldots, 12/16\rangle) = \{\text{Client} : \{\langle 0001, \ldots, 3400\rangle\}, \text{Debt} : \{\langle 0001, 810, 12/7\rangle\}, \text{Loan} : \{\langle 0001, 4000, 12/16\rangle\}\}$. Similarly, $SL_{\bowtie}(\langle 0002, \ldots, 12/19\rangle) = \{\text{Client} : \{\langle 0002, \ldots, 2200\rangle\}, \text{Debt} : \{\langle 0002, 1350, 12/17\rangle\}, \text{Loan} : \{\langle 0002, 20000, 12/19\rangle\}\}$. Then, the source lineage of the both tuples in the join is $SL_{\bowtie}(\{\langle 0001, \ldots, 12/16\rangle, \langle 0002, \ldots, 12/19\rangle\}) = \{\text{Client} : \{\langle 0001, \ldots, 3400\rangle, \langle 0002, \ldots, 2200\rangle\}, \text{Debt} : \{\langle 0001, 810, 12/7\rangle, \langle 0002, 1350, 12/17\rangle\}, \text{Loan} : \{\langle 0001, 4000, 12/16\rangle, \langle 0002, 20000, 12/19\rangle\}\}$.

[2] $\hat{T}_i$ is the subset of tuples in T_i.

$$\left[\left\{\begin{array}{c}\text{Client: } \{\ \langle\ 0002,\ \text{Joa},\ \text{US},\ 2200\ \rangle\ \} \\ \text{Debt: } \{\ \langle\ 0002,\ 1350,\ 12/17\ \rangle\ \} \\ \text{Loan: } \{\ \langle\ 0002,\ 20000,\ 12/19\ \rangle\ \}\end{array}\right\}, \left\{\begin{array}{l}\langle\ \langle\ 2200,\ 1350,\ 20000\ \rangle, \\ \quad \text{Reject},\ \{\text{Debt,LoanAmount}\}\ \rangle\end{array}\right\}\right]$$

Fig. 4 Augmented lineage of tuple set {⟨Reject, 1⟩} in Example 4

Next, we define source lineage in a task. Since a task is an operator tree, the source lineage in the task is obtained by recursively deriving the source lineage in each operator in the operator tree.

Definition 6 (*Source Lineage in Task*) Let $O(= \mathscr{T}(T_1, \ldots, T_m))$ be the result of performing a task $\mathscr{T}$ on the set of tables $\{T_1, \ldots, T_m\}$. Source lineage of tuple $o(\in O)$ in task $\mathscr{T}$ (denoted by $SL_{\mathscr{T}}(o)$) is the set of $\hat{T}_i \subseteq T_i$ which meets the following conditions.

1. If $\mathscr{T}$ contains no operator, namely $O = T_i, SL_{\mathscr{T}}(o) = o \in T_i$.
2. Otherwise, $\mathscr{T}(D) = Op(O_1, \ldots, O_n)$ s.t. $O_i = \mathscr{T}_i(D_i)$, $D_i \subseteq D$. Let $\hat{O}_i = P_{O_i}(SL_{Op}(o))$. Then, $SL_{\mathscr{T}_i}(o) = \bigcup_{1 \le i \le n}(\hat{\bigcup}_{\hat{o_i} \in \hat{O}_i} SL_{\mathscr{T}_i}(\hat{o_i}))$.

The source lineage of tuple set $\bar{O}(\subseteq O)$ in task $\mathscr{T}$ is $SL_{\mathscr{T}}(\bar{O}) = \hat{\bigcup}_{o \in \bar{O}} SL_{\mathscr{T}}(o)$. Note that the source lineage in a task depends on the operator tree and may change even if a different operator tree logically equivalent to the original tree is given, as suggested in [17].

Example 6 (Source lineage in task) Let us consider the partial task $\mathscr{T}$ (π-$\bowtie$-σ) in Fig. 2a, which takes base tables shown in Fig. 3a and produces the table shown in Fig. 3d. Then, the source lineage of tuple (⟨3400, 810, 4000⟩) in the task is $SL_{\mathscr{T}}(\langle 3400,\ 810,\ 4000\rangle) = \{\text{Client} : \{\langle 0001, \ldots, 3400\rangle\}, \text{Debt} : \{\langle 0001,\ 810,\ 12/7\rangle\}, \text{Loan} : \{\langle 0001,\ 4000,\ 12/16\rangle\}\}$.

Before defining the augmented lineage, we define *intermediate lineage* of tuple set $\bar{O}(\subseteq \mathscr{T}(D))$ in an intermediate result. It represents the set of tuples in the intermediate result which contribute to $\bar{O}$.

Definition 7 (*Intermediate Lineage*) Let O be the output tuples of task $\mathscr{T}$. That is, $O = \mathscr{T}(D) = \mathscr{T}(T_1, \ldots, T_m)$. Furthermore, let task $\mathscr{T}$ be divided into two tasks, $\mathscr{T}'$ and $\mathscr{T}''$ s.t. $\mathscr{T}(D) = \mathscr{T}'(O', T_{l_{i+1}}, \ldots, T_{l_m})$, $O' = \mathscr{T}''(T_{l_1}, \ldots, T_{l_i})$. Namely, O' is the intermediate result of the task $\mathscr{T}''$. The intermediate lineage $IL(\bar{O}, O')$ of tuple set $\bar{O}$ $(\subseteq \mathscr{T}(D))$ in intermediate result O' is the source lineage $\hat{O}' \subseteq O'$ of tuple set $\bar{O}$ in task $\mathscr{T}'$, namely $IL(\bar{O}, O') = P_{O'}(SL_{\mathscr{T}'}(\bar{O}))$.

Example 7 (Intermediate lineage) Let $\mathscr{T}$ denote the whole task shown in Fig. 2a, and INT_σ denote the intermediate result of σ in the task shown in Fig. 3b. Then, the intermediate lineage of tuple set {⟨reject, 1⟩} in INT_σ is $IL(\{\langle\text{reject}\rangle\}, \text{INT}_\sigma) = \{\langle 0002,\ 20000,\ 12/19\rangle\}$.

Finally, we define augmented lineage. Augmented lineage consists of source lineage (source tuples contributed to the output tuples) and reasoning lineage (reasons for UDF invocations).

Definition 8 (*Augmented Lineage*) Augmented lineage $AL(\bar{O})$ of tuple set $\bar{O}(\subseteq \mathscr{T}(D))$ in task $\mathscr{T}$ consists of the following pair, where O'_i denotes the intermediate result generated by each function operator $\phi^r_{f_i(\boldsymbol{E}_i)}$ invoked in reasoning mode.

- Source Lineage (SL): $SL_{\mathscr{T}}(\bar{O})$
- Reasoning Lineage (RL): $RL_{\mathscr{T}}(\bar{O}) = \{\langle o.\boldsymbol{E}_i, o.\text{Value}, o.\text{Reason}\rangle | \forall i : o \in IL(\bar{O}, O'_i)\}$

Example 8 (Augmented lineage) Referring to the outputs of the function operator shown in Fig. 3e, the augmented lineage of tuple set {⟨Reject, 1⟩} in the task of Example 4 is shown in Fig. 4.

5 Augmented lineage derivation

As mentioned in Section 2, there are two approaches (i.e., the eager approach and lazy approach) to obtaining the lineage [13,27]. The former produces lineage for all analysis results during the ordinary analysis execution. The latter derives the lineage for the chosen analysis results after the analysis execution. This paper focuses on the lazy approach. In recent data analysis, analysts often develop analysis flows using trial and error to configure parameters, change source data, etc. In this context, the lazy approach is more desirable than the eager approach. The reason is that the lazy one derives the lineage only when needed, while the eager one has larger overhead for every analytical execution even when the lineage is not required. Our proposed method to derive the augmented lineage is based on [17], which proposed a method to obtain lineage of database queries via the lazy approach. Our extension enables use of UDFs for complex data analysis, a more practical treatment of operators and operator trees, and inclusion of reasons for the augmented lineage.

When a tuple in the analysis output is specified as a target for augmented lineage derivation, our method first tries to look for tuples in source tables whose corresponding attribute

values match the attribute values in the specified target tuple as in the lineage derivation in [17]. In cases of simple tasks, this approach directly leads to finding source tuples in the source tables. However, when the task is complex, some attribute values in the target tuple may have been newly generated by operators such as aggregations and functions and do not have directly corresponding attribute values in source tables. In such cases, we need to decompose a task into smaller sub-tasks, and derive augmented lineage step by step using intermediate results of the sub-tasks, rather than deriving augmented lineage from the source tables directly. For this purpose, given a task, we divide its operator tree into one or more operator sub-trees (*segments*). Then, we derive augmented lineage recursively in a top-down manner, as illustrated in the following example.

Example 9 Consider the augmented lineage of a tuple ⟨Reject, 1⟩ in Fig. 1, which is shown in Example 4. To derive its augmented lineage, we divide the analysis into segments because attribute values in the tuple ⟨Reject, 1⟩ are generated by the aggregation and function operators and do not have corresponding attribute values in the source tables. In this example, we have two segments: (1) α and (2) ϕ-π-$\bowtie$-σ. ϕ's intermediate result (namely, the input to α) has five attributes (Income, Debt, LoanAmount, Judge, Reason). Then, we first find the tuple ⟨2200, 1350, 20000, Reject, {Debt, LoanAmount}⟩ as the intermediate lineage of the tuple ⟨Reject, 1⟩ in the intermediate result showing in Fig. 3e, which is the output of ϕ. We also record {⟨⟨2200, 1350, 20000⟩, Reject, {Debt, LoanAmount}⟩} as reasoning lineage. Next, we derive the source lineage of the tuple ⟨2200, 1350, 20000, Reject, {Debt, LoanAmount}⟩ in the source tables. Eventually, we derive the augmented lineage as the pair of source lineage {{⟨0002, Joa, US, 2200⟩}, {⟨0002, 1350, 12/17⟩}, {⟨0002, 20000, 12/19⟩}} and reasoning lineage {⟨⟨2200, 1350, 20000⟩, Reject, {Debt, LoanAmount}⟩}.

In Sects. 5.1 and 5.2, we explain the segment in more detail and how to divide a task into segments, respectively. The segmentation scheme shown here is quite different from those in [17,55] in that it preserves the order of operators in a given operator tree as it is. This feature is beneficial if the given operator tree is optimized by the query optimizer based on the cost model and/or heuristics such as selection push-down. In Sect. 5.3, we introduce a *tracing query*, which finds the source lineage for a single segment. Finally, we propose a basic algorithm to derive the augmented lineage of a task in Sect. 5.4.

5.1 Segment

There are two types of segments (Non-D-segment and D-segment). The original idea of Non-D-segment and D-segment comes from AUSPJ- and D-segments in our reference work [17]. The main reason for this separation will be that due to the non-monotonic property of difference, formulating the tracing query becomes complicated if we mix difference and the other operators. We follow their approach here.

- Non-D-segment: A segment of operators except the difference in the pattern: "ϕ^*-α^*-$(\cup|\pi|\sigma)^*$-$(\pi|\sigma|\bowtie)^*$". ϕ^* and α^* denote sequences of function and aggregation operators, respectively. $(\cup|\pi|\sigma)^*$ and $(\pi|\sigma|\bowtie)^*$ denote the operator sub-trees that consist of any combinations of the specified three operators. The leftmost operator is located at the top of the operator tree. Namely, the operators are executed from right to left in a bottom-up manner. Note that all the operators do not need to actually appear in a Non-D-segment. If α^* consists of multiple aggregations $\alpha_1 - \cdots - \alpha_m$, the grouping keys $\boldsymbol{G_i}$ of α_i must meet the following conditions: $\boldsymbol{G_1} \subseteq \cdots \subseteq \boldsymbol{G_m}$. This segment is based on AUSPJ-segment α-$\cup$-π-σ-$\bowtie$ proposed in [17]. Placing the function operator at the top of the segment naturally extends AUSPJ-segment to accommodate the function operator, since attributes corresponding to Value and Reason of UDFs are guaranteed to appear in its intermediate result, which is convenient for deriving reasoning lineage. Another change is generalization of operator sequences to reduce the number of segments.
- D-segment: A segment consisting of a single difference operator. This segment is represented as the following pattern: "−".

The leftmost operator in a segment is called the *top* of the segment, and the top segment of the whole operator tree is called the *root segment*.

5.2 Segmentation

Given task $\mathcal{T}$, which consists of one or more operators, the *segmentation* splits the task into one or more segments by applying the longest pattern matching to its operator tree in a top-down manner from the root operator node. The patterns here are those explained in the above segment description.

Proposition 1 *The segmentation uniquely decomposes a task into one or more (Non-D/D) segments.*

Proof According to the definitions of segments, any operator can be the top of a segment. Therefore, the segmentation decomposes the operator tree into one or more disjoint segments which covers the whole operator tree. In addition, since we do the longest pattern matching, the decomposition is unique. □

Example 10 The task in Example 4 is divided into two Non-D-segments by the segmentation: (1) Non-D-segment consisting of α alone and (2) Non-D-segment ϕ-π-$\bowtie$-σ (Fig. 2b).

5.3 Tracing query

We define a *tracing query* for each segment. Given a task $\mathscr{T}$ consisting of a single segment, the tracing query finds the source lineage $SL_{\mathscr{T}}(\bar{O})$ of a tuple set $\bar{O}(\subseteq \mathscr{T}(T_1, \ldots, T_m))$ in task $\mathscr{T}$.

First, we introduce the split operator. Given a table and the list of attributes' sets, the split operator projects the table onto each set of attributes.

Definition 9 (*Split operator*) Given a table T with its attribute set $\boldsymbol{A}_T$, the split operator produces a set of tables, each with attribute set $\boldsymbol{A}_{T_i} \subseteq \boldsymbol{A}_T$, using the projection as follows:

$$Split_{\boldsymbol{A}_{T_1},\ldots,\boldsymbol{A}_{T_n}}(T) = \{\pi_{\boldsymbol{A}_{T_1}}(T), \ldots, \pi_{\boldsymbol{A}_{T_n}}(T)\}$$

Example 11 (Split operator) Let $\mathrm{INT}_{\bowtie}$ denote the output table of the join in Fig. 2a shown in Fig. 3c. The split operator $Split_{\boldsymbol{A}_{\mathrm{Client}},\boldsymbol{A}_{\mathrm{Debt}},\boldsymbol{A}_{\mathrm{Loan}}}(\mathrm{INT}_{\bowtie})$ produces {Client : {⟨0001, Havey, UK, 3400⟩, ⟨0002, Joa, US, 2200⟩, ...}, Debt : {⟨0001, 810, 12/7⟩, ⟨0002, 1350, 12/17⟩, ...}, Loan : {⟨0001, 4000, 12/16⟩, ⟨0002, 20000, 12/19⟩, ...}}.

Proposition 2 *A operator tree for a Non-D-segment can be rewritten into the following form (Fig. 5), and the lineage derived for this is same as that derived for the original operator tree.*

$$\mathscr{T}(D) = \phi_{f_1(\boldsymbol{E_1})}(\ldots(\phi_{f_n(\boldsymbol{E_n})}(\alpha_{\boldsymbol{G_1},g_1(\boldsymbol{B_1})} (\ldots(\alpha_{\boldsymbol{G_k},g_k(\boldsymbol{B_k})}(\cup_i(\pi_{\boldsymbol{A_i}}(\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i)))))))))$$

Proof We can prove that this rewriting preserves logical equivalence by the commutativity of operators. As discussed in Sect. 4, rewriting an operator tree may affect the derived lineage. However, it is proved in [17] that this rewriting has no effect on the derived lineage. □

The tracing queries for Non-D-segments and D-segments are shown as follows:

– *Tracing Query for a Non-D-Segment:*
Based on the rewriting in Proposition 2, the source lineage of tuple set $\bar{O}(\subseteq \mathscr{T}(D))$ in task $\mathscr{T}$ can be obtained by executing the following tracing query:

$$TQ_{\bar{O},\mathscr{T}}(D) = \bigcup_i Split_{\boldsymbol{A}_{T_1^i},\ldots,\boldsymbol{A}_{T_{m_i}^i}} (\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i) \ltimes \bar{O})$$

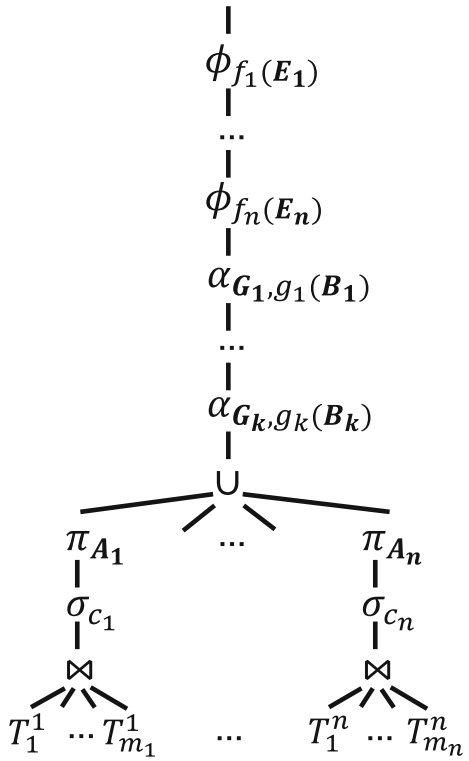


Fig. 5 Transformed non-D-segment

Note that $\boldsymbol{A}_{T_j^i}$ denotes the set of attributes of T_j^i and $\ltimes$ denotes semi-join. If some operators are missing in the Non-D-segment, their counterparts are omitted in the tracing query. If tuple set $\bar{O}$ is the whole output $\mathscr{T}(D)$, the tracing query is denoted as follows using the notation **ALL**:

$$TQ_{\mathbf{ALL},\mathscr{T}}(D) = \bigcup_i Split_{\boldsymbol{A}_{T_1^i},\ldots,\boldsymbol{A}_{T_{m_i}^i}} (\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i))$$

– *Tracing Query for a D-Segment:*
Given a D-segment $\mathscr{T}(D) = T_1 - T_2$, the source lineage of tuple set $\bar{O}(\subseteq \mathscr{T}(D))$ in task $\mathscr{T}$ is represented as follows:

$$TQ_{\bar{O},\mathscr{T}}(T_1, T_2) = \{\bar{O}, T_2\}$$

Please note that the tracing query for a D-Segment identifies T_2 as well as $\bar{O}$ as the source lineage.

5.4 Augmented lineage derivation procedure

We explain an algorithm to derive the augmented lineage of tuple set $\bar{O}(\subseteq \mathscr{T}(D))$ in task $\mathscr{T}$. To obtain the augmented lineage, we (1) split the task into segments as described in Sect. 5.2 and (2) apply Algorithm 1 to the task. Algorithm 1

iteratively applies the tracing query to each segment from the root of the operator tree, derives the source lineage of the segment, and records the reasoning lineage if the tuples in the intermediate result contain reasons.

Example 12 (Augmented lineage derivation procedure) Let us consider the augmented lineage of the tuple ⟨Reject, 1⟩ in the task shown in Example 4 using Algorithm 1. Source data set D consists of three tables (Client, Debt, Loan), the task $\mathscr{T}$ is composed of two segments ($\mathscr{T}_1 = \alpha$ and $\mathscr{T}_2 = \phi$-π-$\bowtie$-σ) as shown in Fig. 2b, and tuple set $\bar{O}$ is {⟨Reject, 1⟩}. Therefore, *TQqueue* is [{⟨Reject, 1⟩}, $\mathscr{T}$] at line 5.

First, [{⟨Reject, 1⟩}, $\mathscr{T}$] is dequeued from *TQqueue* at line 7. Then, since the top segment of the task is Non-D-segment α, we execute the following tracing query at line 28:

$$\begin{aligned}\hat{\text{INT}} &= TQ_{\{\langle \text{Reject}, 1\rangle\}, \mathscr{T}_1}(D) \\ &= \text{INT} \ltimes_{\text{Judge}} \{\langle \text{Reject}, 1\rangle\}\end{aligned}$$

Note that INT denotes the intermediate result of the function operator shown in Fig. 3e, and $\hat{\text{INT}}$ denotes the intermediate lineage of tuple set {⟨Reject, 1⟩} in INT. Then, since $\hat{\text{INT}}$ does not correspond to any source table, we enqueue [$\hat{\text{INT}}$, $\mathscr{T}_2$] into *TQqueue* at line 38. Next, [$\hat{\text{INT}}$, $\mathscr{T}_2$] is dequeued at line 7. Then, we store the reasoning lineage from $\hat{\text{INT}}$ in *RL* at line 24 since $\mathscr{T}_2$ is a Non-D-segment, and its top operator is a function operator. After that, we execute the following tracing query for $\mathscr{T}_2$ at line 28:

$$\begin{aligned}&TQ_{\hat{\text{INT}}, \mathscr{T}_2}(D) = Split_{\boldsymbol{A}_{\text{Client}}, \boldsymbol{A}_{\text{Debt}}, \boldsymbol{A}_{\text{Loan}}} \\ &\quad (\sigma_C(\text{Client} \bowtie \text{Debt} \bowtie \text{Loan}) \ltimes_{\boldsymbol{E}} \hat{\text{INT}}) \\ &\text{s.t. } \boldsymbol{E} = \{\text{Income, Debt, LoanAmount}\} \\ &\quad C : \text{Application date is on December}\end{aligned}$$

Then, since Incôme, Dêbt, and Loân correspond to source tables, they are stored in *SL*. Finally, the algorithm returns the pair of *SL* and *RL* at line 42. Note that $\hat{\text{INT}}$, *SL*, and *RL* are as explained in Example 9.

This algorithm basically assumes that when a task consists of multiple segments, the intermediate result of the lower segment is available for the tracing query to derive the source lineage of the upper segment, and reason information is included in the intermediate result of function operators.[3] We discuss a number of alternative approaches on how to prepare the intermediate results of non-root segments in Sect. 6.

We discuss the computational complexity of the algorithms in terms of two points: (1) the number of executed tracing queries for a task and (2) the complexity of a tracing query itself.

Proposition 3 *(The number of executed tracing queries) The number of executed tracing queries for a task is the same as the number of segments composing the task.*

Proof It is obvious from Algorithm 1. □

Proposition 4 *(Complexity of a tracing query for a Non-D-segment) The tracing query for a Non-D-segment can be represented as follows as discussed in Sect. 5.3.*

$$\begin{aligned}TQ_{\bar{O}, \mathscr{T}}(D) = \bigcup_i &Split_{\boldsymbol{A}_{T_1^i}, \ldots, \boldsymbol{A}_{T_{m_i}^i}} \\ &(\sigma_{C_i}(T_1^i \bowtie \cdots \bowtie T_{m_i}^i) \ltimes \bar{O})\end{aligned}$$

Note that when the tracing query is executed on the intermediate result, $T_1^i \cdots T_{m_i}^i$ represent the intermediate result tables. Let $card(T_j^i)$ denote the cardinality of T_j^i. Then, the complexity of the tracing query is $O(\Sigma_i(\Pi_j\ card(T_j^i) * card(\bar{O})))$.

Proof In the tracing query, joins are dominant operators. For each i, the join/semi-join operation over $m_i + 1$ tables is performed. Thus, we get the above equation. □

Proposition 5 *(Complexity of a tracing query for a D-segment) The complexity of the tracing query for D-segment is $O(1)$.*

Proof It is obvious from the definition of the tracing query for D-segment. □

6 Implementation of augmented lineage derivation

6.1 Deployment of intermediate results

The procedure to derive the augmented lineage shown in Sect. 5.4 assumes that intermediate results of non-root segments are available before execution of tracing queries. In this section, we introduce how to deploy intermediate results.

There are following two naive approaches to do so.

- *Rerun*: This approach runs the original analysis task as it is, usually in the normal mode. When the augmented lineage is requested, it restores all the needed intermediate results of the non-root segments by rerunning the original task in the reasoning mode. Although it causes no runtime overhead and storage cost to the original analysis task, rerunning the task to restore the intermediate results takes much time.

[3] When there is a D-segment ($O = O_1 - O_2$), the algorithm does not need intermediate results O_1. If O_2 is the result of a Non-D-segment without the function operator, it does not need O_2, either.

Algorithm 1 Augmented Lineage Derivation Procedure

Input: Source data set D, Task $\mathscr{T}$, Tuple set $\bar{O}$ ($\subseteq O = \mathscr{T}(D)$)
Output: Augmented lineage of tuple set $\bar{O}$ in task $\mathscr{T}$

$SL = RL = \emptyset$;
// *TQqueue* is a queue.
// $[T_1, \ldots, T_n]$ means a list consisting of T_i.
Initialize *TQqueue*;
Enqueue $[\bar{O}, \mathscr{T}]$ into *TQqueue*;
while *TQqueue* is not empty **do**
  Dequeue $[\bar{O}, \mathscr{T}]$ from *TQqueue*;
  if $\mathscr{T}'s$ top segment $\bar{\mathscr{T}}$ is a D-segment **then**
    // $O = \bar{\mathscr{T}}(O_1, O_2) = O_1 - O_2$
    // $O_i = \mathscr{T}_i(D_i)$ s.t. $D_i \subseteq D$
    if $\bar{O} =$ **ALL then**
      $\bar{O} \Leftarrow O$;
    end if
    $[\hat{O}_1, \hat{O}_2] \Leftarrow [\bar{O}, \textbf{ALL}]$;
  else if $\mathscr{T}'s$ top segment $\bar{\mathscr{T}}$ is a Non-D-segment **then**
    // $O = \bar{\mathscr{T}}(O_1, \ldots, O_k)$
    // $O_i = \mathscr{T}_i(D_i)$ s.t. $D_i \subseteq D$
    if $\bar{\mathscr{T}}'s$ top operators are $\phi_{f_1(\boldsymbol{E_1})}$-...-$\phi_{f_l(\boldsymbol{E_l})}$ **then**
      if $\bar{O} =$ **ALL then**
        $\bar{O} \Leftarrow O$;
      end if
      for $o \in \bar{O}$ **do**
        for $idx = 1, \ldots, l$ **do**
          $RL = RL \cup \{\langle o.\boldsymbol{E_{idx}}, o.\text{Value}_{idx}, o.\text{Reason}_{idx}\rangle\}$;
        end for
      end for
    end if
    $[\hat{O}_1, \ldots, \hat{O}_k] \Leftarrow TQ_{\bar{O},\bar{\mathscr{T}}}([O_1, \ldots, O_k])$;
  end if
  for Each $\hat{O}_i$ **do**
    if $\hat{O}_i$ corresponds to source table T_i **then**
      if $\hat{O}_i =$ **ALL then**
        $\hat{O}_i$ is replaced with the set of tuples in T_i.
      end if
      // $\hat{O}_i$ is added as a member corresponding to T_i.
      $SL = SL \cup \{\hat{O}_i\}$;
    else
      Enqueue $[\hat{O}_i, \mathscr{T}_i]$ into *TQqueue*;
    end if
  end for
end while
return $[SL, RL]$;

– *Full Materialization (Full)*: This approach executes the original analysis task in the reasoning mode and materializes (generates and stores) the needed intermediate results of non-root segments during its execution. This approach affects the performance of the original task and needs the storage cost for managing the intermediate results.

In the preliminary experiments, we evaluated the performance of the above two approaches on complex data analyses. We found that the execution cost of function operator with an expensive UDF tends to be dominant in the overall processing time compared with other relational operators.

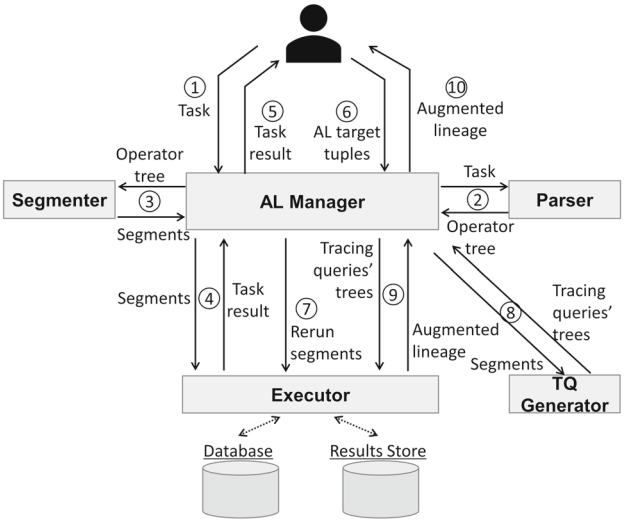


Fig. 6 System architecture

Hence, based on this observation, we propose the following alternative approach to providing the intermediate results of non-root segments.

– *Function Materialization (FM):* This approach executes the original analysis task in the reasoning mode as in Full Materialization but materializes only intermediate results of non-root segments which contain function operators during its execution. Intermediate results of the other segments are restored by rerunning the segments.

To reduce the time of rerunning tasks and segments in Rerun and FM, we utilize the semi-join pushdown optimization used in [57]. When we get output tuples O of a task and $\bar{O} \subseteq O$ are specified as targets for the augmented lineage derivation, we apply the semi-join $O \ltimes \bar{O}$ and then push it down along the operator tree. Although [57] does not consider the difference operator, the pushdown transformation $((T_1 - T_2) \ltimes \bar{O} \rightarrow (T_1 \ltimes \bar{O}) - T_2)$ is applicable. This optimization will contribute to reducing the size of tables accessed in rerunning the analysis. We experimentally evaluate the cost of the above approaches.

6.2 System architecture

This section describes the system architecture which enables complex data analysis with augmented lineage. The overall organization is shown in Fig. 6. The system consists of five components: AL Manager, Parser, Segmenter, TQ Generator, and Executor.

AL Manager: It receives tasks from the user, manages the translation to operator trees, their segmentation and execution, and returns the task results to the user. When the user requests the augmented lineage for some result tuples, it man-

ages the tracing query generation and execution, and returns the augmented lineage to the user.

Parser: It takes a task from AL Manager and translates it into an optimized operator tree.

Segmenter: It takes an operator tree from AL Manager and decomposes it into segments.

TQ Generator: It takes a segment of a task and generates its tracing query in the form of an operator tree.

Executor: Given operator tree(s), it executes the tree(s) over Database and Results Store, and returns the results. Results Store preserves the task output and intermediate results needed for augmented lineage derivation.

We explain the procedure of the system to derive augmented lineage assuming Function Materialization. Rerun and Full Materialization can be accommodated with minor adjustment. In the following procedure, Steps ①–⑤ correspond to the task execution, and Steps ⑥–⑩ correspond to the augmented lineage derivation.

① The user submits the task to AL Manager.
② AL Manager passes the given task to Parser and obtains its operator tree.
③ AL Manager sends the tree to Segmenter and receives its segments.
④ The segments are passed to Executor. Executor executes the trees while storing necessary intermediate results in Results Store. AL Manager obtains the task result.
⑤ AL Manager returns the task result to the user.
⑥ The user specifies the target tuples for augmented lineage derivation (AL target tuples) to AL Manager.
⑦ AL Manager identifies segments needed to rerun for the intermediate result deployment and passes them to Executor. Executor executes them and store their results in Results Store.
⑧ AL Manager sends all the segments of the task to TQ Generator, and TQ Generator returns an operator tree of the tracing query for each segment.
⑨ AL Manager performs Algorithm 1 with the help of Executor and obtains augmented lineage.
⑩ AL Manager returns the augmented lineage to the user.

7 Experiment

This section experimentally evaluates the performance of three methods to derive augmented lineage. We use two datasets (LFW datasets [28] and Complex Data Analysis workload (CDA workload)). In the CDA workload, we used the extended TPC-H benchmark queries [50] including complex content processing involving ML-based image and text analysis. For each dataset, we first evaluate the processing time of executing analysis tasks. In the case of Full Materialization and Function Materialization, the system needs to store intermediate results while executing analysis tasks. This intermediate result store takes an extra cost over the ordinary analysis execution. We evaluate this additional cost along with the cost for executing analysis tasks. Then, we evaluate the three methods regarding processing time and storage cost for deriving augmented lineage. As described in Sect. 6.1, we utilize the semi-join pushdown optimization proposed in [57] to rerun the task efficiently. Therefore, the Rerun approach here is compatible with their framework.

For this experiment, we developed a prototype which has a simplified version of the architecture discussed in Sect. 6.2. The prototype implements AL Manager as python scripts which also integrate functionality of Segmenter and TQ Generator. PostgreSQL is deployed to accommodate Database and Results Store and to provide functionality of Parser and Executor as well. Thus, we employ SQL to specify segments and tracing queries to be evaluated in Executor. We implemented this prototype using PostgreSQL 9.6 and Python 3.7.8, and we ran it on a machine with an Intel(R) Core(TM) i7-8700 CPU @ 3.20 GHz, a GeForce GTX 1060 3 GB, and two 16 GB DDR4 DIMMs.

We rebooted the machine before each measurement, and all measurements were independently performed 5–20 times. We show the number of measurements in each experiment section and the average value as the result.

The bar graphs below suggest statistical differences with ** (significance level 1%) and * (significance level 5%) under Welch's *t*-test [51]. We omitted error bars representing confidence intervals, since their intervals are too narrow to see.

7.1 LFW dataset evaluation

In this experiment, we evaluate the performance of the three methods using LFW dataset and investigate the impact of the processing cost of the function operator in the whole task. We prepared two UDFs which have different processing costs. All the results here are based on ten times measurement.

7.1.1 Task

We use the LFW dataset [28] for this experiment. The database consists of two tables: Image and Event, whose schemas are shown in Appendix. The attribute *i_img* stores one photograph image from the LFW dataset, and it actually contains the path (URI) of the external photograph image file as a string. The path includes the name of the celebrity in the photograph. The attribute *e_visitors* stores the number of participants in the event. The other attributes store identifiers for images, events, and places as integer values. Values for attributes except for *i_img* were synthesized so that they maintain consistency.

The SQL for the analysis task in this experiment is also shown in Appendix. It uses UDF for person recognition for

Table 1 Number of tuples in each source table

	Small	Medium	Large
Image	4.5×10^4	4.5×10^5	4.5×10^6
Event	6.075×10^5	6.075×10^6	6.075×10^7

images and determines the number of times that celebrities appeared on stage at large event places. We prepared two implementations of the *recognition* UDF function with difference processing costs. Each implementation is explained in detail in Sect. 7.1.2.

We prepared three different source table sizes (Small, Medium, Large). Table 1 shows the number of tuples in each case.

7.1.2 UDFs

The *recognition* UDF takes the path to the photograph image file as an argument and returns the person's name in the photograph. We prepared two different implementations as follows:

- Face Recognition
  A person is identified by an ML-based face recognition model in the given image. It outputs the person's name as Value and the position of the bounding box around the face as Reason. Its processing cost is more expensive than the following alternative.
- String Processing
  A person is recognized by extracting the substring corresponding to the celebrity's name from the URI string. It outputs the person's name as Value and the position of the name substring in the URI as Reason. Its processing cost is cheaper than the face recognition implementation.

Note that the output value of the two *recognition* UDF implementations is the same.

7.1.3 Task processing

This subsection compares the processing time of the task (analysis query) corresponding to the three methods (Rerun, FM, and Full). In the case of Rerun, we execute the analysis query as it is, since Rerun needs no stored intermediate results. In the cases of FM and Full, we need to store some intermediate results while executing the analysis query. Full needs more storage than FM.

We used the medium-sized tables for this experiment. Figure 7 shows the result. The left-hand side shows the processing time of the analysis query involving the expensive UDF, and the right-hand side shows the processing time for

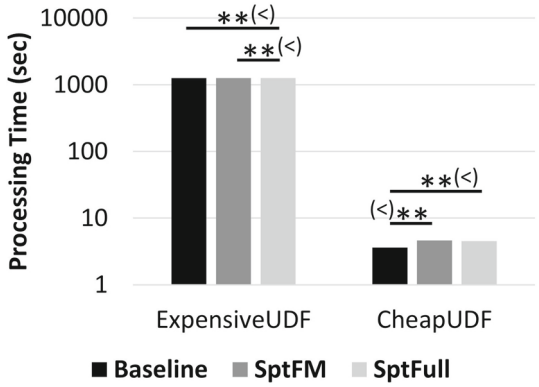


Fig. 7 Task processing time

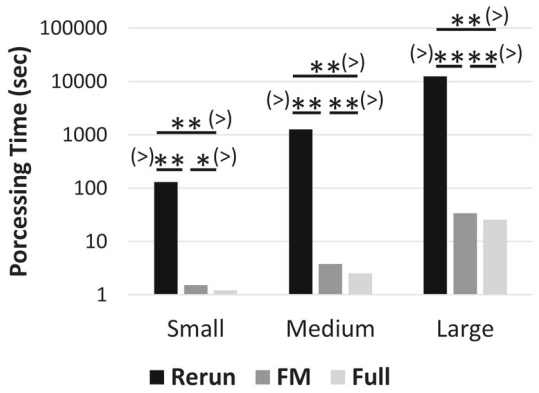


Fig. 8 Processing time for deriving the augmented lineage in the analysis with the expensive UDF

the cheap UDF. In the figure, Baseline corresponds to executing the analysis query without storing any intermediate results for Rerun, and SptFM and SptFull correspond to the processing time including the storage of intermediate results for FM and Full, respectively.

As shown in the figure, we validated the trend that SptFM and SptFull incur some processing overhead compared with Baseline, and the latter needs more. The reason is that SptFull stores more intermediate results. In addition, the increase in processing overhead is larger in the case of the cheap UDF, since the storage cost of intermediate results is almost same no matter whether the UDF is expensive or cheap.

7.1.4 Derivation of augmented lineage

In this subsection, we measure the processing time to derive augmented lineage for a single tuple in the analysis query results.

Face recognition UDF (expensive UDF) case: Figure 8 summarizes the processing time. Table 2 shows the number of tuples in the stored intermediate results. Both Function Materialization (FM) and Full Materialization (Full) outperform Rerun with respect to the processing time. FM is up to

Table 2 Number of tuples in the stored intermediate results

	Small	Medium	Large
FM	4.5×10^3	4.5×10^4	4.5×10^5
Full	4.95×10^4	4.95×10^5	4.95×10^6

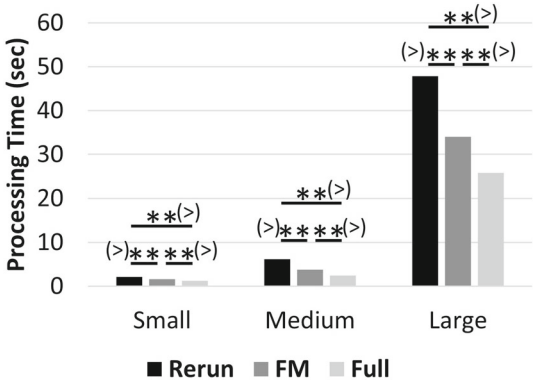


Fig. 9 Processing time for deriving the augmented lineage in the analysis with the cheap UDF

371.9 times faster than Rerun for the large-sized tables, and Full is up to 497.1 times faster than Rerun for the medium-sized tables. FM drastically reduces the number of stored intermediate tuples (by 90.9%) compared to Full (Table 2).

String processing UDF (cheap UDF) case: Figure 9 summarizes the processing time. In this case, FM and Full also exceed Rerun with respect to the processing time. FM is 1.6 times faster than Rerun, and Full is 2.6 times faster than Rerun for the medium-sized tables. Since the number of tuples in the stored intermediate results is the same as in the expensive UDF case, FM incurs a smaller storage cost than Full. However, the advantage of storing the intermediate results on the processing time is smaller because the processing cost of UDF is not so expensive compared with other relational operators. Therefore, the gain of FM and Full over Rerun has become smaller.

The above experiments demonstrate that FM can control the tradeoff between the processing time for augmented lineage derivation and the storage cost of the intermediate results.

7.2 CDA workload evaluation

This section measures the performance of the three methods using the CDA workload.[4] Since original TPC-H [18,50] is composed of 22 SQL queries focusing on evaluating the performance of a relational DBMS, it originally does not involve any complex data analysis. We extended source tables by adding and replacing some attributes, developed new scenarios for complex data analysis, and extended the original queries to include UDFs whose processing costs are comparable to that of the expensive UDF in the LFW Dataset experiment.

We explain database extension and analysis queries in Sect. 7.2.1, and UDFs in Sect. 7.2.2. Then, we show the processing time for executing each analysis query in Sect. 7.2.3. Finally, we report the processing time for deriving augmented lineage in Sect. 7.2.4.

7.2.1 Tasks

We extended the original TPC-H database. The main extension is as follows:

- Part table
  We added a new attribute to store the photograph image of the part (*p_image*). We used images in the COCO dataset [34] for this extension, and we stored path strings of the image files as its values.
- Lineitem table
  We replaced the comment attribute (*l_comment*) values with Amazon review text data [40].

Since we extended the database as described above, we can develop new complex data analysis scenarios on the original queries. For the Part table, we added filtering that selects parts whose photograph images include any human image based on the human detection UDF. For the Lineitem table, we added another filtering that selects line-item tuples whose review comment is positive based on the text sentiment classifier UDF.

We picked up 10 queries (*Q2*, *Q3*, *Q9*, *Q15*, *Q16*, *Q17*, *Q18*, *Q19*, *Q20*, and *Q21*) from the 22 TPC-H benchmark queries for the CDA workload. These queries were chosen by the following procedure: We first chose 19 TPC-H queries which refer to the Part and/or Lineitem tables. Next, we filtered out 3 queries which contain the CASE clause because our model does not cover it. Then, we applied the following two changes to the remaining 16 queries:

1. Rewrite queries containing nested subqueries (e.g., IN clause and EXISTS clause) by join and difference operators.
2. Add the UDFs to do the above content processing.

Among the 16 queries, Full Materialization and Function Materialization happen to do the same processing for 10 queries to derive augmented lineage. We chose 4 representative queries among them. Eventually, we selected 10 queries,

[4] This workload is derived from the TPC-H Benchmark and is not comparable to published TPC-H Benchmark results, as this implementation does not comply with the TPC-H Benchmark.

Table 3 CDA workload queries

CDA workload query	Original TPC-H benchmark query	Changes	Involved segment
Q1	Q2	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image	Non-D
Q2	Q3	Apply *text_sentiment* UDF to Lineitem table and select line-item tuples whose review comments are positive	Non-D
Q3	Q9	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image	Non-D
Q4	Q15	Apply *text_sentiment* UDF to Lineitem table and select line-item tuples whose review comments are positive	Non-D
Q5	Q16	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image	Non-D
Q6	Q17	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image	Non-D
Q7	Q18	Apply *text_sentiment* UDF to Lineitem table and select line-item tuples whose review comments are positive	Non-D
Q8	Q19	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image	Non-D
Q9	Q20	Apply *person_detection* UDF to Part table and select parts whose photograph images include any human image. Then, add distinct clause to remove duplicate	Non-D
Q10	Q21	Apply *text_sentiment* UDF to Lineitem table and select line-item tuples whose review comments are positive	Non-D and D

the 4 queries plus 6 queries in which Full Materialization and Function Materialization do different processing.

We summarize the derivations of the 10 queries from original queries and the types of involved segments in Table 3. Note that *person_detection* is the UDF applied to the attribute *p_image*, and *text_sentiment* is applied to the attribute *l_comment*. Finally, we input those SQL queries to PostgreSQL and obtained processing trees output by the EXPLAIN command. Then, by applying our segmentation algorithm, we decided segments to be used for tracing queries. Hereafter, **Qxx** denotes the query xx of CDA workload.

In this experiment, we generated databases of three sizes using DBGEN [18,50], whose scale factor (SF) is 1, 10 and 100.

7.2.2 UDFs

We implemented the above two UDFs. Both of the UDFs utilize a neural network for their prediction.

- Human detection
  This UDF is applied to the attribute *p_image* in the extended Part table. This UDF takes image data as input and decides whether any human image appears in the whole image or not. It returns a Boolean value as Value and the position of the bounding box of the human image as Reason when it returns True. We implemented this UDF using YOLO [44], which can detect objects quickly.
- Text sentiment classifier
  This UDF is applied to the attribute *l_comment* in the extended Lineitem table. This UDF takes review comment text data as input and returns Positive or Negative, representing the sentiment of the review comment, as Value and likelihood of the sentiment as Reason. We implemented this UDF using BERT-Base text sentiment classifier provided by IBM [29].

7.2.3 Task processing

Figure 10 shows the processing time for analysis queries. This result is based on ten times measurements. Note that SptFM and SptFull do the same processing for **Q2**, **Q3**, **Q5**, and **Q8**. This shows that the overhead of storing intermediate results while executing the analysis is negligible. The average overhead of SptFM and SptFull for all queries is 0.9 % and 0.5 %, respectively. This experiment suggests that the overhead of storing intermediate results while analysis query execution is rather small when the UDF processing cost is not negligible.

7.2.4 Derivation of augmented lineage

In this experiment, we evaluate the performance when we derive augmented lineage for target output tuples (AL_size).

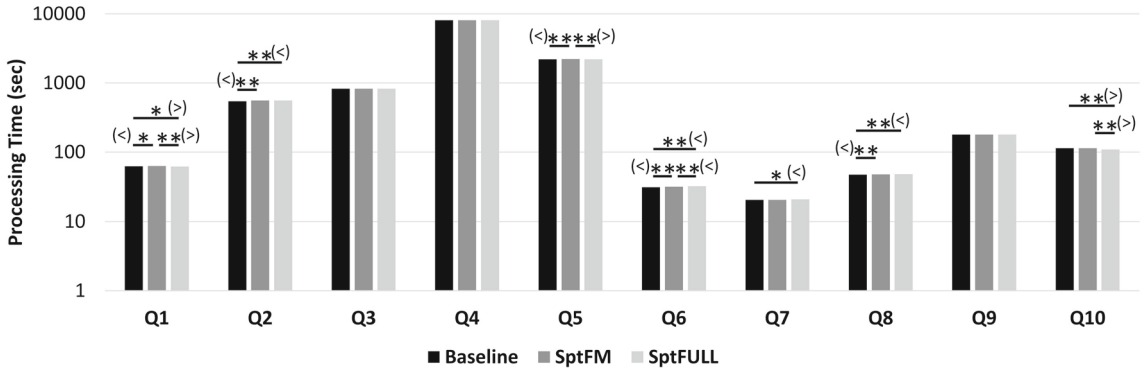


Fig. 10 Task processing time

First, for all queries, we measure the processing time for deriving augmented lineage for a randomly chosen single tuple in the analysis query results. **Q1**, **Q3**, **Q5**, **Q9**, and **Q10** have more than 100 output tuples. Then, we also measure the processing time for deriving augmented lineage for randomly chosen 100 tuples for these queries. The results for SF = 1, 10, 100 are based on 20, 10, 5 times measurements, respectively.

The processing time to derive the augmented lineage for a single output tuple (AL_size = 1) for tables of the three different scale factors is shown in Fig. 11a–c, and that for 100 output tuples is shown in Fig. 12a–c. We set the timeout to 24 h, which means that we stopped deriving augmented lineage when it would take more than 24 h. The processing time of Rerun for **Q3** and **Q4** is not shown due to the timeout in Figs. 11c and 12c. Table 4 shows the number of tuples in the stored intermediate results. Note that FM and Full do the same processing for **Q2**, **Q3**, **Q5**, and **Q8**.

In the case of AL_size = 1: We can see the trend that (1) Rerun is the most time-consuming, (2) Function Materialization (FM) is faster than Rerun, and (3) Full Materialization (Full) is the fastest for deriving augmented lineage. Namely, Rerun > FM > Full. However, FM often achieves the processing time comparable to Full. Regarding the storage cost, FM needs less stored tuples than Full as shown Table 4.

However, we can see some different cases, too. The first one is the case of Full > Rerun > FM. This case applies to **Q7** in Fig. 11b and c. In the query, Rerun and FM could rerun the analysis query quickly due to the semi-join and predefined indexes for source tables. Moreover, since the intermediate results obtained by the rerun were far smaller than the stored intermediate results in Full and cached in the DBMS, they could execute the tracing queries quickly. On the other hand, Full needed to scan the huge intermediate results without using index. This will be the reasons why Rerun and FM outperformed Full in this query. In **Q5** in Fig. 11c, FM and Full do the same processing, and Rerun outperformed FM and Full. In addition to the above mentioned reason, we found that the query plan of a tracing query was poor in FM and Full.

In some queries, Rerun > FM/Full. **Q1** in Fig. 11a and b, **Q4** in Fig. 11a–c, and **Q9** in Fig. 11a are such cases. In **Q7** in Fig. 11a, Rerun > Full > FM, and in **Q1** in Fig. 11c, Full > Rerun/FM. In these cases, FM was accelerated due to the similar reason as mentioned above.

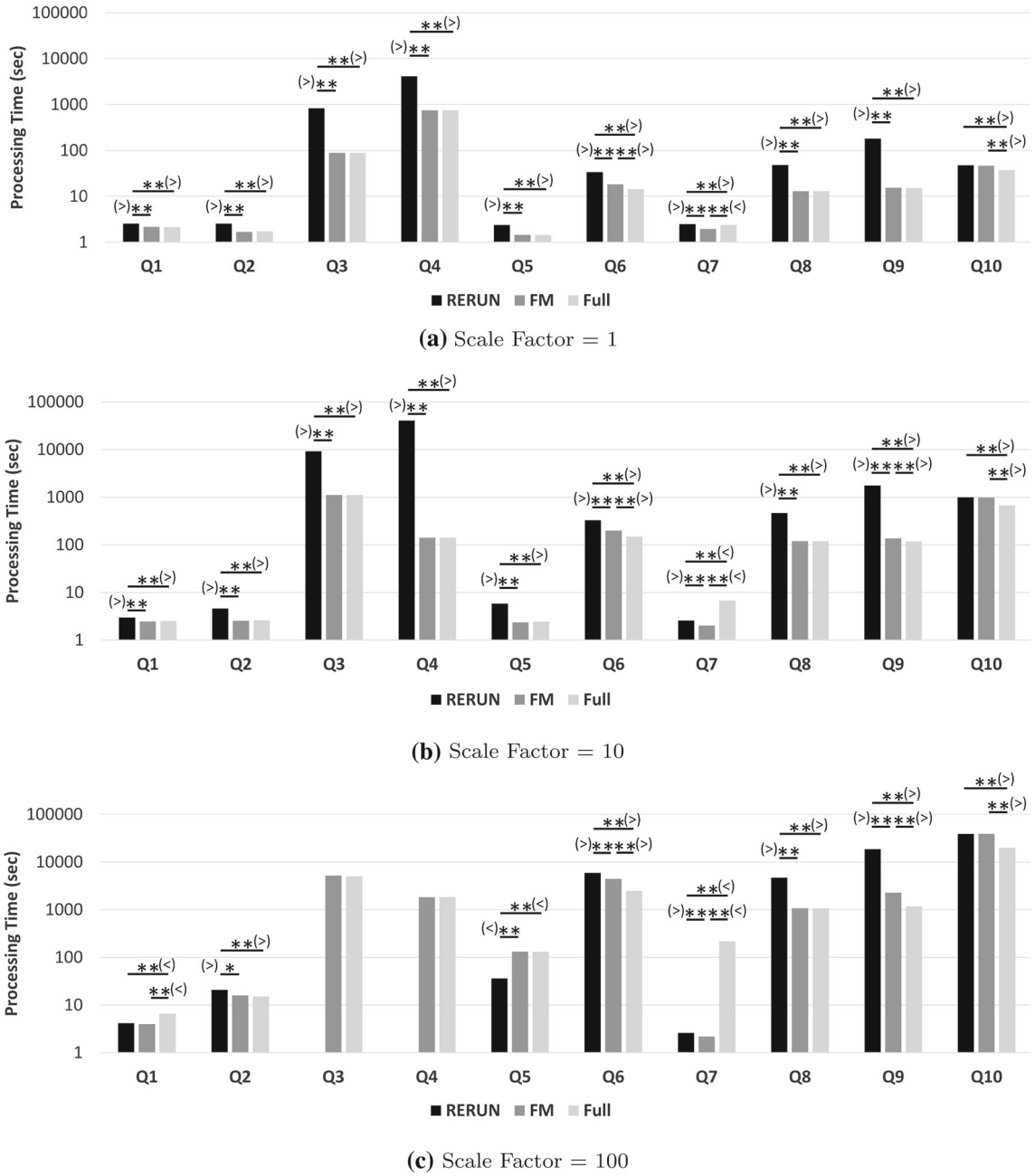
In **Q10** in Fig. 11a–c, Rerun/FM > Full because the cost of rerunning the UDFs was negligible.

We can see that the processing time generally increases as the scale factor increases. In **Q1** and **Q7**, Rerun and FM took almost the same time even when the scale factor increased. In these queries, the processing time of the analysis queries was not affected much because of the efficient semi-join. Besides, since the size of the intermediate result restored by rerunning was almost same for the three scale factors, the processing time of the tracing query in Rerun and FM was not affected.

In **Q4**, the processing time of FM and Full decreased even though the scale factor was increased from 1 to 10. The reason was that the query plans of the tracing query were different for SF = 1 and SF = 10, and that for SF = 1 was inefficient. In **Q5**, the processing time of FM and Full suddenly increased over Rerun due to the similar reason. Namely, the query plan of the tracing query for SF = 100 was poor in FM and Full.

In the case of AL_size = 100: When AL_size increases, the overall processing time generally increases. The increase in the processing time is $x1$–$x13$ in most queries, and the maximum is $x26$ in Rerun for **Q5** in SF = 100. Rerunning the analysis query takes more time since the semi-join produces more source tuples. In addition, the tracing query needs to process larger input tables. The comparison of the three approaches is almost the same as in AL_size = 1.

In **Q10**, Rerun and FM outperformed Full in SF = 1. **Q10** needs four tracing queries (TQ1, TQ2, TQ3, and TQ4) to get the augmented lineage. In SF = 10 and SF = 100, the execution of TQ4 was dominant in Full, while in SF = 1, the execution of TQ3 accounted for almost half of the whole processing time. Rerun and FM could run TQ3 very quickly using the cache. This will be the reason for **Q10** in SF = 1.



(a) Scale Factor = 1

(b) Scale Factor = 10

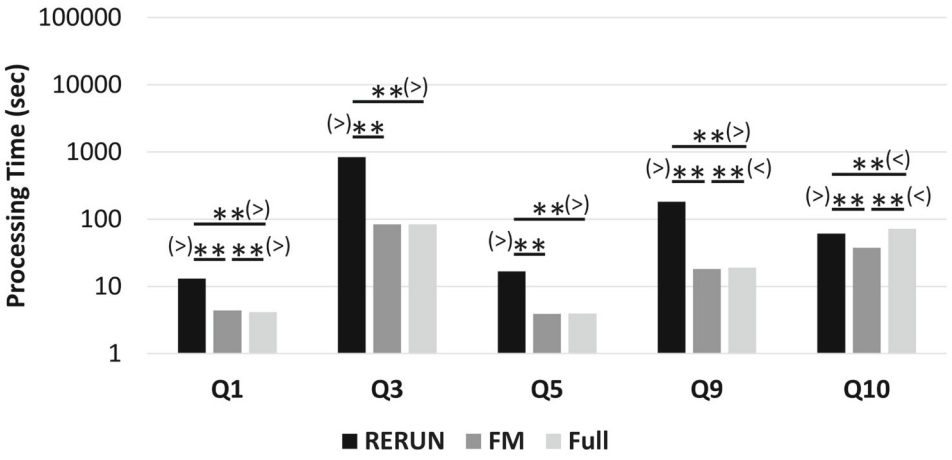
(c) Scale Factor = 100

Fig. 11 Processing time for deriving augmented lineage in the case of AL_size = 1

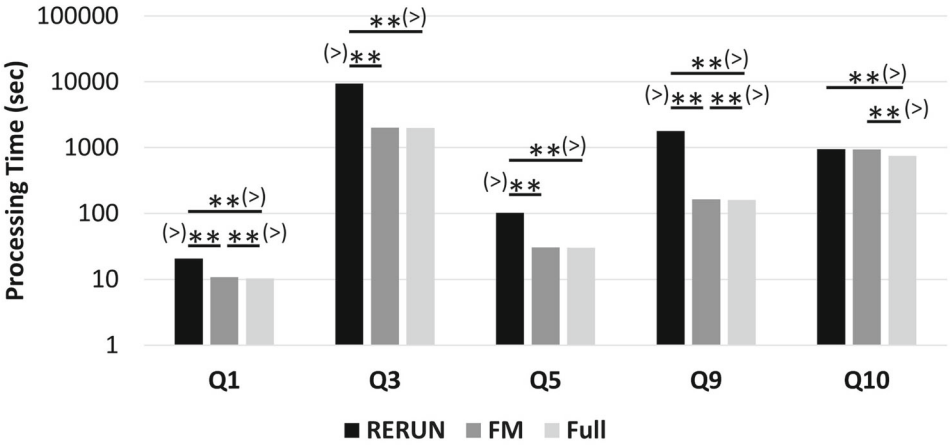
These experiments demonstrate that FM is an appropriate approach to realize a tradeoff between processing time and storage cost. For example, in **Q9**, FM can derive augmented lineage 12.8 times faster than Rerun and reduce the number of stored intermediate tuples by 99.6% in the case of SF = 10 and AL_size = 1.

8 Conclusions and future work

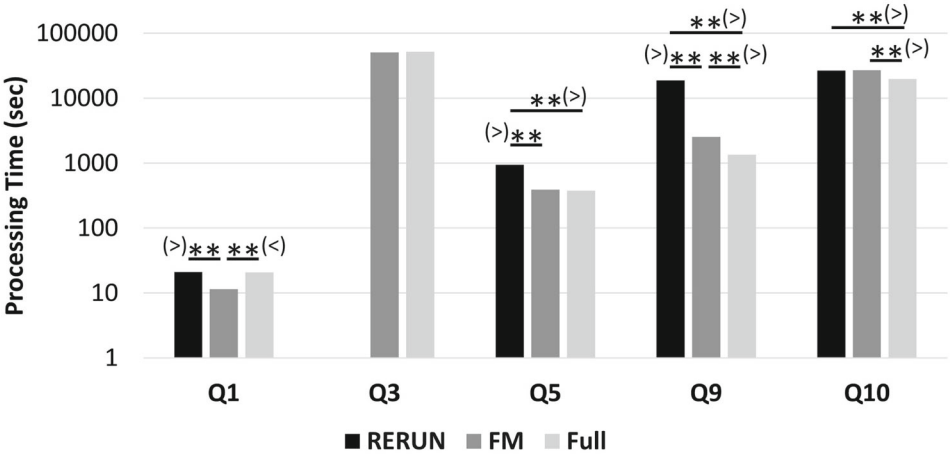
In this paper, we have proposed the augmented lineage, which is an extended lineage combining reasons for complex data analysis. Augmented lineage ensures traceability of complex data analysis including UDFs for AI/ML processing. Additionally, we formulated an algorithm to derive the augmented lineage using the lazy approach. We also proposed a new execution scheme named Function Materialization (FM), which allows for a tradeoff between runtime cost and



(a) Scale Factor = 1



(b) Scale Factor = 10



(c) Scale Factor = 100

Fig. 12 Processing time for deriving augmented lineage in the case of AL_size = 100

Table 4 Number of tuples in the stored intermediate results

	FM	Full	FM	Full	FM	Full
	Q1		Q2		Q3	
SF1	7.89×10^2	1.17×10^5	3.02×10^4	3.02×10^4	1.09×10^4	1.09×10^4
SF10	7.92×10^3	1.18×10^6	3.00×10^5	3.00×10^5	1.08×10^5	1.08×10^5
SF100	8.00×10^4	1.19×10^7	2.99×10^6	2.99×10^6	1.09×10^6	1.09×10^6
	Q4		Q5		Q6	
SF1	4.60×10^5	4.70×10^5	2.96×10^4	2.96×10^4	2.05×10^2	2.00×10^5
SF10	4.59×10^6	4.69×10^6	2.96×10^5	2.96×10^5	2.05×10^3	2.00×10^6
SF100	4.59×10^7	4.69×10^7	2.97×10^6	2.97×10^6	2.01×10^4	2.00×10^7
	Q7		Q8		Q9	
SF1	6.30×10^1	1.50×10^6	4.71×10^2	4.71×10^2	2.22×10^3	5.47×10^5
SF10	5.88×10^2	1.50×10^7	4.70×10^3	4.70×10^3	2.17×10^4	5.47×10^6
SF100	5.12×10^3	1.50×10^8	4.81×10^4	4.81×10^4	2.18×10^5	5.48×10^7
	Q10					
SF1	3.78×10^3	7.55×10^3				
SF10	3.97×10^4	7.95×10^4				
SF100	3.99×10^5	7.98×10^5				

storage cost in deriving the augmented lineage. Experiments showed that FM is effective, especially when the execution time of UDFs involving sophisticated AI/ML processing is high in the whole analysis tasks.

Interesting future research topics include generalization of our reasoning framework. This paper assumes that UDFs performing complex data analysis like AI/ML processing provide reasons. Since the reason is a kind of annotation, it could be used for other purposes, too. For example, we may use it to show an operator's processing time or resource utilization to monitor the system behavior. In addition, extending the proposed framework to more general data models and more generalized analysis contexts such as big data processing systems and stream processing environments is also an interesting issue. Finally, integrating model-agonistic explanation systems (e.g., LIME [45] and SHAP [38]) into our framework is also a challenging topic.

Acknowledgements This work was partly supported by JSPS KAKENHI Grant Numbers JP19H04114 and JP22K19802, NEDO Grant Number JPNP20006, AMED Grant Number JP21zf0127005, and JST SPRING Grant Number JPMJSP2124.

Appendix: LFW dataset evaluation

We show the database schema and analysis query for the LFW dataset evaluation. The underlined attributes denote the primary key of the table. Note that *recognition* is the UDF applied to the attribute i_img. We highlight the part applying UDF using the underline.

Database schema

- Image($\underline{i\_imageid}, i\_placeid, i\_img$)
- Event($\underline{e\_eventid}, e\_placeid, e\_visitors$)

Analysis query

```
SELECT i_value, COUNT(*)
FROM (
  SELECT i_imageid, i_placeid, avg, i_img,
(recognition(i_img)).value i_value
  FROM Image, (
      SELECT e_placeid, avg(e_visitors) avg
      FROM Event
      GROUP BY e_placeid
  ) Seg2
  WHERE i_placeid = e_placeid
    AND avg >= 50000
) Seg1
GROUP BY i_value;
```

References

1. Ainy, E., Bourhis, P., Davidson, S.B., Deutch, D., Milo, T.: Approximated summarization of data provenance. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM '15, pp. 483–492. Association for Computing Machinery, New York, NY, USA (2015)
2. Akoush, S., Sohan, R., Hopper, A.: Hadoopprov: towards provenance as a first class citizen in mapreduce. In: 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13). USENIX Association, Lombard, IL (2013)
3. Amsterdamer, Y., Deutch, D., Tannen, V.: Provenance for aggregate queries. In: Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '11, pp. 153–164 (2011)
4. Benjelloun, O., Das Sarma, A., Halevy, A., Theobald, M., Widom, J.: Databases with uncertainty and lineage. VLDB J. **17**(2), 243–264 (2008)
5. Benjelloun, O., Sarma, A.D., Halevy, A., Widom, J.: ULDBs: Databases with uncertainty and lineage. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, pp. 953–964 (2006)
6. Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An annotation management system for relational databases. VLDB J. **14**(4), 373–396 (2005)
7. Biton, O., Cohen-Boulakia, S., Davidson, S.B., Hara, C.S.: Querying and managing provenance through user views in scientific workflows. In: 2008 IEEE 24th International Conference on Data Engineering, pp. 1072–1081 (2008)
8. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: a survey. ACM Comput. Surv. **37**(1), 1–28 (2005)
9. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: a characterization of data provenance. In: Database Theory—ICDT 2001, pp. 316–330 (2001)
10. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Vistrails: Visualization meets data management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, pp. 745–747. Association for Computing Machinery, New York, NY, USA (2006)
11. Chalapathy, R., Chawla, S.: Deep learning for anomaly detection: a survey. arXiv:1901.03407 (2019)
12. Cheney, J.: A formal framework for provenance security. In: 2011 IEEE 24th Computer Security Foundations Symposium, pp. 281–293 (2011)
13. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in Databases: Why, How, and Where. Now Publishers Inc (2009)
14. Chiticariu, L., Tan, W.C., Vijayvargiya, G.: Dbnotes: A post-it system for relational databases based on provenance. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, pp. 942–944 (2005)
15. Cui, Y., Widom, J.: Practical lineage tracing in data warehouses. In: Proceedings of 16th International Conference on Data Engineering, pp. 367–378 (2000)
16. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. VLDB J. **12**(1), 41–58 (2003)
17. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. **25**(2), 179–227 (2000)
18. Deister software: TPCH benchmark. https://docs.deistercloud.com/content/Databases.30/TPCH%20Benchmark.90
19. Deutch, D., Gilad, A., Moskovitch, Y.: Selective provenance for datalog programs using top-k queries. Proc. VLDB Endow. **8**(12), 1394–1405 (2015)
20. Deutch, D., Milo, T., Roy, S., Tannen, V.: Circuits for datalog provenance. In: Proceedings of the 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24–28, 2014, pp. 201–212 (2014)
21. Du, M., Liu, N., Hu, X.: Techniques for interpretable machine learning. Commun. ACM **63**(1), 68–77 (2019)
22. Foster, J.N., Green, T.J., Tannen, V.: Annotated xml: queries and provenance. In: Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '08, pp. 271–280. Association for Computing Machinery, New York, NY, USA (2008)
23. Glavic, B., Alonso, G.: Perm: processing provenance and data on the same data model through query rewriting. In: 2009 IEEE 25th International Conference on Data Engineering, pp. 174–185 (2009)
24. Goecks, J., Nekrutenko, A., Taylor, J.: Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. **11**(8), 1–13 (2010)
25. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 31–40 (2007)
26. Gunning, D.: Explainable Artificial Intelligence (XAI). Defense Advanced Research Projects Agency (DARPA) (2017)
27. Herschel, M., Diestelkämper, R., Ben Lahmar, H.: A survey on provenance: What for? What form? What from? VLDB J. **26**(6), 881–906 (2017)
28. Huang, G.B., Ramesh, M., Berg, T., Learned-Miller, E.: Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Reports 07-49, University of Massachusetts, Amherst (2007)
29. IBM: Text Sentiment Classifier—IBM Developer. https://developer.ibm.com/exchanges/models/all/max-text-sentiment-classifier/
30. Interlandi, M., Shah, K., Tetali, S.D., Gulzar, M.A., Yoo, S., Kim, M., Millstein, T., Condie, T.: Titian: Data provenance support in spark. In: Proceedings of the VLDB Endowment International Conference on Very Large Data Bases, vol. 9, pp. 216–227. NIH Public Access (2015)
31. Islam, S.R., Eberle, W., Ghafoor, S.K., Ahmed, M.: Explainable artificial intelligence approaches: A survey. CoRR arXiv:2101.09429 (2021)
32. Kermany, D.S., Goldbaum, M., Cai, W., Valentim, C.C., Liang, H., Baxter, S.L., McKeown, A., Yang, G., Wu, X., Yan, F., et al.: Identifying medical diagnoses and treatable diseases by image-based deep learning. Cell **172**(5), 1122–1131 (2018)
33. Kwon, D., Kim, H., Kim, J., Suh, S.C., Kim, I., Kim, K.J.: A survey of deep learning-based network anomaly detection. Clust. Comput. **22**(1), 949–961 (2019)
34. Lin, T.Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C.L., Dollár, P.: Microsoft coco: Common objects in context. arXiv:1405.0312 (2014)
35. Litjens, G., Sánchez, C.I., Timofeeva, N., Hermsen, M., Nagtegaal, I., Kovacs, I., Hulsbergen-Van-De, Kaa C., Bult, P., Van Ginneken, B., Van Der Laak, J.: Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis. Sci. Rep. **6**(1), 1–11 (2016)
36. Logothetis, D., De, S., Yocum, K.: Scalable lineage capture for debugging disc analytics. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13. Association for Computing Machinery, New York, NY, USA (2013)
37. Lucia, B., Ceze, L.: Data provenance tracking for concurrent programs. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 146–156 (2015)
38. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. Adv. Neural Inf. Process. Syst. **30**, 4765–4774 (2017)
39. Ma, S., Aafer, Y., Xu, Z., Lee, W.C., Zhai, J., Liu, Y., Zhang, X.: Lamp: Data provenance for graph based machine learning

algorithms through derivative computation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 786–797 (2017)
40. Ni, J., Li, J., McAuley, J.: Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 188–197 (2019)
41. Palyvos-Giannas, D., Gulisano, V., Papatriantafilou, M.: GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. Parallel Comput. **89**, 102–552 (2019)
42. Palyvos-Giannas, D., Havers, B., Papatriantafilou, M., Gulisano, V.: Ananke: a streaming framework for live forward provenance. Proc. VLDB Endow. **14**(3), 391–403 (2020)
43. Psallidas, F., Wu, E.: Smoke: fine-grained lineage at interactive speed. Proc. VLDB Endow. **11**(6), 719–732 (2018)
44. Redmon, J., Farhadi, A.: Yolov3: an incremental improvement. arXiv:1804.02767 (2018)
45. Ribeiro, M.T., Singh, S., Guestrin, C.: "Why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1135–1144 (2016)
46. Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D.: Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 618–626 (2017)
47. Senellart, P.: Provenance and probabilities in relational databases. SIGMOD Rec. **46**(4), 5–15 (2018)
48. Souza, R., Azevedo, L.G., Lourenço, V., Soares, E., Thiago, R., Brandão, R., Civitarese, D., Vital-Brazil, E., Moreno, M., Valduriez, P., Mattoso, M., Cerqueira, R., Netto, M.A.S.: Workflow provenance in the lifecycle of scientific machine learning. Concurr. Comput. Pract. Exp. **34**(14), e6544 (2022)
49. Theoharis Y., Fundulaki I., Karvounarakis G., Christophides V.: On provenance of queries on semantic web data. IEEE Internet Comput. **15**(1), 31–39 (2011)
50. Transaction Processing Performance Council: TPC-H Homepage. http://tpc.org/tpch/
51. Welch, B.L.: The generalization of 'student's' problem when several different population varlances are involved. Biometrika **34**(1–2), 28–35 (1947)
52. Woodruff, A., Stonebraker, M.: Supporting fine-grained data lineage in a database visualization environment. In: Proceedings 13th International Conference on Data Engineering, pp. 91–102 (1997)
53. Wu, E., Madden, S., Stonebraker, M.: Subzero: a fine-grained lineage system for scientific databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 865–876 (2013)
54. Wu, Y., Tannen, V., Davidson, S.B.: Priu: A provenance-based approach for incrementally updating regression models. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, pp. 447–462 (2020)
55. Yamada, M., Kitagawa, H., Amagasa, T., Matono, A.: Augmented Lineage: Traceability of Data Analysis Including Complex UDFs. In: Database and Expert Systems Applications, pp. 65–77. Springer International Publishing (2021)
56. Zhang, S., Yao, L., Sun, A., Tay, Y.: Deep learning based recommender system: a survey and new perspectives. ACM Comput. Surv. (CSUR) **52**(1), 1–38 (2019)
57. Zheng, N., Alawini, A., Ives, Z.G.: Fine-grained provenance for matching and etl. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 184–195 (2019)