Tracing nested data with structural provenance for big data analytics

Ralf Diestelkämper Melanie Herschel
IPVS - University of Stuttgart
Stuttgart, Germany
ralf.diestelkaemper|melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

Big data analytics systems such as Apache Spark natively support nested data formats since they offer operators to manipulate nested lists and complex types. Compared to flat data, nested data introduces further complexity and sources of error, e.g., when developing data processing pipelines, performing auditing tasks, or performance tuning. To ease such tasks, we propose a provenance-based solution tailored to nested data processing in big data analytics systems. Unlike previous solutions, it combines (i) tracing provenance of *nested data* with (ii) *efficient* and *scalable* provenance processing, leveraging a newly proposed *structural provenance* that traces structural manipulations through data processing pipelines in addition to data. We provide a formal definition of structural provenance, as well as methods to efficiently capture and succinctly backtrace it. We implement them in our Pebble system in Apache Spark and validate its performance and usefulness on up to 500GB of real-world data.

1 MOTIVATION

Big data analytics systems such as Apache Spark or Flink are frequently the means of choice to build data processing pipelines that process large quantities of nested data. These pipelines transform nested lists and complex types stored in nested data formats like JSON, protocol buffer, or parquet. Provenance solutions that capture meta-data about the data processing [14] have proven to be useful for analyzing the internals of data processing pipelines, e.g., for debugging purposes. These solutions typically have two phases, a provenance capture phase to collect the meta-data and a provenance query phase to analyze the meta-data. For big data analytics systems, we distinguish two categories of provenance solutions: (i) Efficient and scalable solutions that track individual, flat data items (i.e., tuples) from the input to the output over each execution step [15–17, 22]. They capture so-called lineage or why-provenance [7]. (ii) System prototypes that compute provenance polynomials of nested data [2, 28]. They capture how-provenance, which provides both the input items contributing to the result and the data combination process an item undergoes.

Solutions of the first category fail to track nested items accurately. Solutions of the second category do not efficiently scale to big data processing pipelines. To capture the how-provenance, these solutions propagate the growing provenance polynomial through the entire pipeline or require annotation of each nested element, which imposes a very high and practically unacceptable overhead [16, 17]. Further, these solutions have to offload the provenance to external tools to query the captured provenance. Thereby, they miss potential performance and usability benefits compared to solutions that are fully integrated into the big data analytics or data-intensive, scalable computing (DISC) system.

© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

We, therefore, present a *DISC system integrated* provenance solution for nested data that is as *efficient* and *scalable* as solutions of the first category while, at the same time, at least as *accurate* as solutions of the second category [9]. Our solution leverages our newly defined *structural provenance* to provide both features.

Structural provenance records identifiers for top-level data items only. For attributes and nested data items, it captures paths on a schema level. To provide accurate provenance when queried, it employs these identifiers and paths to trace back individual nested items at attribute level. Capturing paths instead of identifier annotations for nested data further allows us to distinguish between paths that are used for access (e.g., during filtering) or manipulation (e.g., during flattening). We can thereby differentiate contributing attributes, i.e., attributes needed to reproduce a result item, and influencing attributes that are accessed during data processing but not required to reproduce a result. This distinction, which is unique compared to existing data provenance models, qualifies structural provenance for use-cases beyond debugging such as auditing or determining data-usage patterns for partitioning, data compression, and workload optimization.

Auditing. Auditing aims at identifying and analyzing data breaches. These breaches commonly stem from attacks of company insiders who extract sensitive data by querying data and leaking the query result. Auditing solutions are designed to identify both these insiders and the customers whose data are leaked [19]. To address the latter challenge, the solutions typically leverage some sort of data provenance. It serves to identify those input tuples that are exposed in a leaked query result. However, after the European Union has introduced the European general data protection regulation *GDPR* [26], European companies are not only required to identify the customers (tuples) whose data are leaked, but also which of their data are leaked (i.e., attributes such as name, address, or payment details). Structural provenance precisely provides the attributes and items in nested collections that contribute to a query result. Unlike existing data provenance solutions, it further reveals which attributes are not exposed in the result but have influenced it to create awareness for reconstruction attacks.

Data-usage patterns. Data-usage patterns reveal frequently used subsets of the input data over a query workload. These patterns serve to optimize data layout and compression or to improve query performance [25]. State-of-the-art scalable provenance solutions for DISC systems can identify subsets of the input data that are frequently used. This knowledge allows for horizontal (or row-based) data partitioning and distribution. Structural provenance further provides all the information needed for vertical (or column-based) partitioning since it reveals which attributes and nested items are accessed or manipulated. It even provides insights on attribute combinations that are frequently used together for data layout optimizations.

Capturing provenance imposes runtime and space overhead during pipeline execution. The mentioned use-cases are performed infrequently. Thus, keeping the overhead low during pipeline execution is essential to ensure efficiency and scalability. During the *provenance capture* phase, a system can typically opt for computing and storing the provenance of all processed data (*eager* approach) or decide to capture it on demand when users query the provenance (*lazy* approach). Consequently, during the *provenance query* phase, retrieving the desired provenance is more or less time-consuming. We consider the provenance capture and provenance query phases holistically. To this end, we devise a meet-in-the-middle approach that eagerly collects the necessary "pebbles" (i.e., identifiers and paths on schema level) during pipeline execution to later reconstruct or backtrace attribute-level provenance of nested data at query time. Our evaluation shows that capturing structural provenance introduces comparable overhead to state-of-the-art lineage solutions in DISC systems [17], while providing attribute-level precision.

This paper also presents the first provenance solution for nested data that seamlessly integrates into a big data analytics system (Apache Spark in our implementation). Existing solutions [2, 28] require offloading captured provenance for querying to separate, non-distributed applications. This has three drawbacks: (i) It prevents adopting a holistic provenance capture and querying approach to keep capture and query overhead reasonable; (ii) it forces users to leave their familiar environment; and (iii) it prevents scalable provenance querying.

Contributions and structure. To summarize, this paper presents research on processing structural provenance in big data analytics systems to accurately trace nested data in an efficient, scalable, and integrated way. This approach enables novel use-cases that arise in the context of big data processing. After discussing a running example in Sec. 2 and related work (Sec. 3) this paper covers the following contributions:

- **Structural provenance (Sec. 4).** We present a novel provenance model for nested data that tracks structural manipulations in addition to data dependencies, and distinguishes between data access and manipulation to support use-cases beyond debugging.
- **Lightweight structural provenance capture (Sec. 5).** We discuss how to capture structural provenance in big data analytics programs composed of filter, select, map, join, union, flatten, grouping, nesting, and aggregation operations. The capture is devised to incur a minimal overhead compared to the capture of flat provenance in DISC systems.
- **Backtracing for provenance query processing (Sec. 6).** We formalize the backtracing algorithm used at provenance query time. As input, users provide a tree-pattern that, upon its scalable execution, identifies data items for which provenance is requested. The backtracing algorithm computes provenance for these items based on the previously captured information.
- **Implementation and evaluation (Sec. 7).** We implement our contributions in Pebble [9], our system for integrated provenance capturing and querying within Apache Spark. We conduct a quantitative evaluation of runtime and space overhead incurred by our solution on two large real-world data sets, validating the scalability of our solution. In comparison to the state-of-the-art lineage solution Titian [17], Pebble has comparable runtime and space overhead. However, as our workload shows, Pebble provides sufficient insight to support the above use-cases, unlike other solutions.

2 RUNNING EXAMPLE

To distinguish our research from related work and for illustration, we use a running example based on Twitter data. Among its roughly 1000 attributes, we focus on the tweeted *text*, the *user* tweeting, the *user_mentions* in the tweet, and the *retweet_cnt*. The input data is nested as shown in Tab. 1 (ignore colors and number annotations for now). This sample data is processed in the big data processing pipeline shown in Fig. 1. It results in a list of distinct users associated with tweets that they authored or were mentioned in, as shown in Tab. 2. The upper branch of the pipeline describes how authoring users become part of the result. Their tweets require a *retweet_cnt* of 0 before the pipeline reduces them to the *text*, *id_str*, and *name*. The lower branch processes tweets mentioning users. First, it flattens the *user_mentions* attribute to select the tweeted *text*, *id_str*, and *name* of each mentioned user. Then the pipeline unifies the results of both branches and groups by the user to aggregate the tweeted *texts* into a nested list.

In the result, a duplicate *Hello World* text occurs in the nested *tweets* of user *Lisa Paul*, short *lp*. To find out how this potential data quality issue occurred, we debug the pipeline by tracing back the duplicate texts in the context of user *lp*, which are highlighted in dark-green in Tab. 2. The solution presented in this paper returns the dark- and medium-green items in Tab. 1. The dark-green items are contributing data. They suffice to reproduce the dark-green items in the result. The medium-green items reveal which attributes potentially influence the result of the pipeline.

If trivially extended to nested data, scalable lineage solutions [15–17, 22] provide all input tweets that contain the user *lp*. They are highlighted in light-grey in the input. In reality, a user typically authors more than a handful of tweets and is potentially mentioned in more than a million tweets. These tweets would all be in the provenance returned by the lineage solutions. They mask the actual two tweets causing the duplicate text.

PROVision [28] supports the unnesting of data but does not explicitly support the nesting of data. Extending it with nesting requires a list collection UDF *cl*, which yields the following provenance polynomial for the entire result item 102 in Tab. 2:

$$(p_1+p_{12}+p_{17}+(p_{29}\cdot P_{flatten}(p_{29}\cdot[0])))\cdot$$
$$P_{cl}((p_1+p_{12}+p_{17}+(p_{29}\cdot P_{flatten}(p_{29}\cdot[0]))),(\langle p_1\rangle+\langle p_{12}\rangle+\langle p_{17}\rangle+\langle(p_{29}\cdot P_{flatten}(p_{29}\cdot[0]))\rangle))$$

	text	user		user_mentions		retweet_cnt
		id_str	name	id_str	name	
1	Hello @ls @jm @ls[2]	lp[3]	Lisa Paul[4]	ls[5] jm[7] ls[9]	Lauren Smith[6] John Miller[8] Lauren Smith[10]	0[11]
12	Hello World[13]	lp[14]	Lisa Paul[15]			0[16]
17	Hello World[18]	lp[19]	Lisa Paul[20]			0[21]
22	This is me @jm[23]	jm[24]	John Miller[25]	jm[26]	John Miller[27]	0[28]
29	Hello @lp[30]	jm[31]	John Miller[32]	lp[33]	Lisa Paul[34]	1[35]

Table 1: Example input data

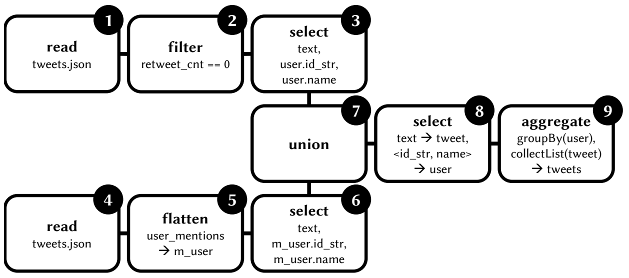


Figure 1: Example processing pipeline

	user		tweets
	id_str	name	text
101	ls	Lauren Smith	Hello @ls @jm @ls Hello @ls @jm @ls
102	lp	Lisa Paul	Hello @ls @jm @ls Hello World Hello World Hello @lp
103	jm	John Miller	Hello @ls @jm @ls This is me @jm This is me @jm

Table 2: Example result data

Essentially, the first line tells us that the result item is based on the source tuples annotated with 1, 12, and 17, denoted as p_1, as well as p_{12}, p_{17} (all these are processed by the upper branch of the pipeline in Fig. 1), and p_{29}, with some of its data flattened out during processing (corresponds to the lower part of the pipeline). The second line makes use of our extension and describes how data is combined by the remainder of the pipeline where the tuples mentioned above are grouped and aggregated. The example shows that the provenance is very verbose while not precisely tracing the dark-green data items of the user question. This is the case since it collects tuple-based provenance polynomials only.

Lipstick [2] traces provenance polynomials for each nested item. This allows pinpointing the dark-green nested values *Hello World* and *lp* correctly. However, Lipstick requires annotating all values, not just the tuples, e.g., 35 rather than 5 annotations, as indicated by the superscript italic numbers in Tab. 1. This entails a significant runtime and space overhead, rendering the solution impractical when needing to scale to large volumes of data.

We also differentiate structural provenance from where-provenance [4], which determines where a (nested) result value is copied from. In our example, the where-provenance (extended to the processing pipelines we consider) would include, for the value *lp* the "cells" with superscript annotation 14, 19, and 33 of Tab. 1. This is combined with the where-provenance of the *Hello World* result values via product. The result is not sufficiently accurate because it cannot capture that the dark-green values of the output need to be traced within their common context.

No existing solution allows recognizing (i) that the *user* attribute is unnested and nested again, (ii) that the *id_str*, *lp*, and the *text* attribute *Hello World* are subject to different, independent, structural manipulations, and (iii) that the medium-green *retweet_cnt* and *name* values in Tab. 1 are accessed for filtering and grouping, respectively. Even though these values are not needed to reproduce the queried result, they are influencing the result, which is valuable information in certain use-cases. Structural provenance captures all this information since it captures not only data dependencies but also path dependencies.

To get an understanding of querying structural provenance, consider the right tree in Fig. 2. The string labels of tree nodes denote attribute names whereas numbers refer to provenance ids (e.g., 102) or positions in nested collections (e.g., 2 and 3). The displayed tree represents the structure associated with our sample user query. It encodes the path to user *lp* and the duplicate *Hello World* items in the context of top-level data item 102. Note that *name* is absent from this tree since it is not pertinent to the user query. Backtracing this tree yields the two trees on the left of Fig. 2. These distinguish between data items that contribute to the result (dark-green) and data items that influence it (medium-green). The nodes match the green items in Tab. 1. A closer look at the medium-green *name* node reveals that this node influences the queried result since it is accessed for grouping

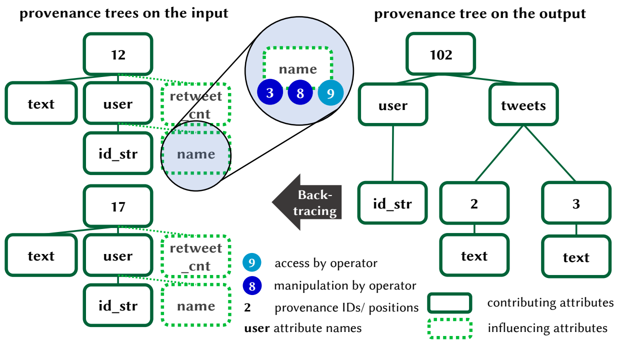


Figure 2: Example provenance trees. Tracing the tree on the right back to the input yields the trees on the left

(light-blue 9). Similarly, the *retweet_cnt* influences the result since it is accessed for filtering. Further, the *name* node undergoes structural manipulations at operator 3 and 8 (dark-blue).

3 RELATED WORK

This section generalizes the discussion of existing approaches that we provided along with the running example. We divide our discussion into research on data provenance in DISC systems and provenance models for nested data, summarized in Tab. 3.

3.1 Data provenance in DISC systems

Data provenance has been studied for various applications [14]. While the majority of approaches has focused on relational data processed by relational queries, first solutions have emerged for tracing provenance in DISC systems such as Titian [12, 16, 17] for Spark, Lipstick for PigLatin (Hadoop) [2], as well as RAMP [15], Newt [22], and PROVision [28] for multiple DISC systems.

Titian, RAMP, and Newt trace lineage of data items, i.e., they determine which top-level data items contribute to which output item. These solutions scale well but do not trivially extend to nested data. PROVision extends the provenance model for top-level data (or flat) items to also capture provenance of data items in nested collections. It lacks information on attribute level access. Lipstick is the only solution that supports provenance capture for nested data at attribute level. However, it requires annotations for each data value, not only the top-level data items. Structural provenance provides provenance on attribute level but requires annotation on top-level items only since it records access to attributes and nested data using paths. These paths are recorded on a schema level, saving space and runtime overhead.

All the above solutions except for Titian require offloading the provenance to an external tool. Titian integrates provenance querying directly into the DISC system. Thus, provenance queries can be integrated into a big data processing pipeline just like any other query. Our system extends Titian's integrated querying means with tree-patterns [13, 23] to address combinations of nested data items. Further, we present the first solution that tracks access and manipulation of attributes.

3.2 Provenance models for nested data

Focusing on nested data, at least three major directions to formalize provenance models have been researched: (i) models for why-, how-, and where-provenance, (ii) graph-based provenance models, and (iii) program slicing models.

For unions of conjunctive queries, Buneman et al. [4] define a why- and where-provenance model for nested data. This model

Feature	Titian	Ramp	Newt	Lipstick	PROVision	HowProvNested	Why/Where Prov	Kwasnikowa	Acar	Program Slicing	Structural Prov
Data provenance for nested data	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Provenance of acces and manipulation	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Provenance of data item structure	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Eager/lazy provenance computation	✓ / ✗	✓ / ✗	✓ / ✗	✓ / ✗	✗ / ✓	n.a.	n.a.	n.a.	✗ / ✓	✓ / ✓	✓ / ✓
Implementation-independent provenance query formalism	✗	✗	✗	✓	✗	n.a.	n.a.	na.	✗	✗	✓
DISC system compatibility/integration	✓ / ✓	✓ / ✗	✓ / ✗	✓ / ✗	✓ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✓ / ✓
Reported implementation	Spark RDDs	Hadoop	Hadoop/ Hyracks	PigLatin	Java	no	no	no	Haskell	Haskell	Spark Datasets
Evaluated for scalability	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓

✓ Supported ✗ Not Supported

Table 3: Feature overview of related work

does not extend to the programs defining data analytics pipelines in DISC systems, like the one shown in Fig. 1, since they may include map or reduce functions or any other higher-order functions in general. To model the how-provenance of nested data, a semiring-model for a subset of XQuery has been proposed [10, 18]. However, this model does not include complex operations over nested data, such as aggregations. The only how-provenance model supporting aggregations that we are aware of applies to relational data only [3].

Lipstick [2], makes use of a graph model to describe the how-provenance. This model only applies semiring annotations where possible. It provides no formal model definition for aggregations, nesting, and flattening of nested data. Kwasnikowska and Acar et al. [1, 20] also employ a graph-based provenance model to track nested data items. These solutions are essentially limited to the operations defined in the Nested Relational Calculus (NRC) [5], which do not include aggregations or joins. Also, the reported implementation and evaluation (if any) indicate that they neither integrate nor scale sufficiently to apply on DISC systems.

All provenance solutions mentioned so far do not distinguish between access and manipulation as they focus on tracing data values. In that respect, the work closest to our structural provenance model is the program slicing model [6], which tracks provenance traces for NRC operators over nested data. To provide formal guarantees, the model is limited to a small set of semantically fully specified NRC operators. In practice, it is infeasible to provide semantics for all higher-order functions such as map operations, which allow for user-defined functions. Via trace slicing it is possible to query provenance for individual nested items. However, like the other described models, this model is designed to trace data values and manipulations of them rather than structural manipulations. It is not expressive enough to faithfully capture and query structural manipulations. Its implementation is not designed or evaluated for efficiency or scalability.

The final column of Tab. 3 summarizes the capabilities of our system, which we have highlighted previously. These capabilities are based on processing structural provenance, discussed next.

4 STRUCTURAL PROVENANCE

This section formalizes structural provenance. We first present the data model and the execution model to define the corresponding structural provenance model afterwards.

4.1 Data model

DISC systems process collections of typed nested data items, which we refer to as (nested) datasets. These datasets support positional access, and, thus, the handling of ordered datasets.

Definition 4.1. (*Nested dataset*) A nested dataset D comprises constants, data items, bags, and sets, denoted and typed as shown in Tab. 4. D is a list of data items $d_1, \ldots, d_n$ with or without duplicates (ordered bag vs. set), i.e., $D = B|S$. Each data item d is

Name	Notation	Type $\tau(\cdot)$
Constant	c	$Int\|Double\|String\|\ldots$
Data item	$d = \langle a_1 : v_1, \ldots, a_n : v_n\rangle$	$\langle a_1 : \tau(v_1), \ldots, a_n : \tau(v_n))\rangle$
Bag	$B = \{\{d_1, \ldots, d_n\}\}$	$\{\{\tau(d)\}\}, \forall d, d' \in B, \tau(d) = \tau(d')$
Set	$S = \{d_1, \ldots, d_n\} \|d_1 \neq \ldots \neq d_n$	$\{\tau(d)\}, \forall d, d' \in S, \tau(d) = \tau(d')$

Table 4: Notation and types for nested collections

a list of $a_i : v_i$ pairs. Attribute names $a_1, \ldots, a_n$ are unique labels within each data item. Values $v_1, \ldots, v_n$ may be bags, sets, data items, or constants, i.e., $v = B|S|c|d$.

The type of D is defined recursively based on the type of its building blocks as described in Tab. 4, where $\tau(\cdot)$ returns the type of its parameter. Bags and sets are restricted to containing elements of the same type.

Example 4.2. All data shown in our running example conform to the above definition. The result data of Tab. 2 has type:
$\{\{\langle user{:}\langle id\_str{:}String, name{:}String\rangle, tweets{:}\{\{\langle text{:}String\rangle\}\}\rangle\}\}$

To access the different components defined by the data model, we define *access paths*, inspired by XPath expressions [24] to navigate XML data. Provided a context data item d, an access path navigates to "deeper" data in the nested model. Given that the data model ensures the order of data items in lists, we also model positional accesses in paths.

Definition 4.3. (*Access path w.r.t. d*) In the context of a data item d, we define an access path p by $p = d.p'$, $p' = x \mid x.p'$, $x = a \mid a[i]$. Here, p' is the path accessing x either directly or recursively. The accessed x is either an attribute a in the schema of the context data item, evaluating to its value, or the i-th component of a, denoted $a[i]$, evaluating to the item at the i-th position of a's value. For the recursive definition of p', the context data item is updated to the item referred to by x.

For simplification, we denote a path p with context data item d by p^d when the context is not clear. We refer to the enumeration of all paths that exist in a context d as path set PS^d.

Example 4.4. Considering the data item d_{102} in Fig. 2, the path $d_{102}.tweets$ evaluates to a list of four data items. Path $d_{102}.tweets[2].text$ points to the first *Hello World* in that list.

4.2 Execution model

The execution model defines the processing semantics of data analytics programs like the one in Fig. 1. These programs process data complying with our data model. We model a program as a directed acyclic graph (DAG) of individual operators, such as filter, flatten, join, etc. Each operator has its own execution semantics.

Definition 4.5. (*Operator*) An operator O takes a set of datasets $I = \{I_1, \ldots, I_k\}$ as input and returns a single result dataset R. Inference rules describe the execution semantics of an operator O. O has a unique identifier, a type, and its arguments.

Definition 4.6. (*Program execution model*) Let $G(V, E)$ be a DAG. $V = \{O_1, \ldots, O_n\}$ is the set of algebraic operators and E the set

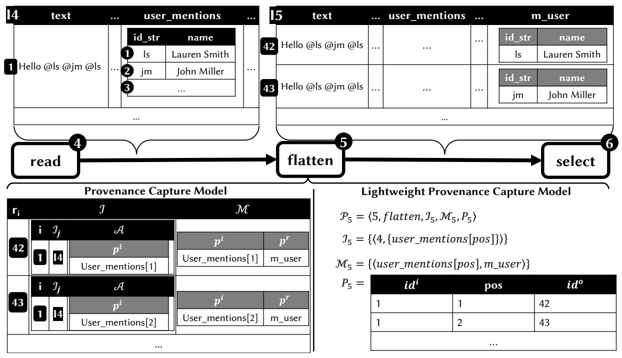


Figure 3: Provenance model and lightweight provenance capture applied on the flatten operator from the example

of directed edges that model the data flow. If and only if the result set R_i of O_i is in the set I_j of input datasets of operator O_j an edge $(O_i, O_j) \in E$ directed from O_i to O_j exists. While G may contain multiple source nodes (in-degree of 0), G has only one sink node (out-degree of 0), which outputs the final result dataset.

Example 4.7. The graph in Fig. 1 represents the execution model of our running example. The type and parameters of each operator are displayed inside the operator nodes. Further, each node is labeled with its identifier in the top right corner.

We abstract from a particular language to define the semantics of individual operators by extending the inference rules from [11] to describe the filter, select, map, join, union, flatten, grouping and aggregation operator with their semantics.

Example 4.8. We illustrate how inference rules work on our inference rule for a *join* operator:

$$\frac{\varphi(i,j) \Rightarrow \text{true}}{I_1.join[\varphi(i \in I_1, j \in I_2)](I_2) \Rightarrow \{\{\langle i,j\rangle \mid i \in I_1, j \in I_2\}\}}$$

The rule joins two input datasets I_1 and I_2 into a single result dataset. More precisely, the operator associates elements $i \in I_1$ with elements $j \in I_2$ based on a join condition ($\varphi(i,j) \rightarrow boolean$). With the precondition that φ evaluates to true, i.e., $\varphi(i,j) \Rightarrow$ true, the data item $\langle i,j\rangle$ becomes part of the result.

4.3 Provenance model

Our provenance model extends the program execution model described above by adding annotations to each node in G. More precisely, for each operator O represented by a node in G, it generates the result provenance $\mathcal{R}$ that contains the provenance of each result data item r_i in the result R of operator O.

Definition 4.9. (Result provenance w.r.t. result R of operator O) Let $\mathcal{R} = \{\rho_1, \ldots, \rho_n\}$ be the result provenance associated with $R = \{r_1, \ldots, r_n\}$. For each data item $r_i \in R$, we record the result data item provenance $\rho_i = \langle r_i, \mathcal{I}, \mathcal{M}\rangle$, where $\mathcal{I}$ is the provenance of input data items that contribute to r_i (see definition below) and $\mathcal{M}$ is a set of path pairs mapping access paths of input data items to paths of r_i to describe restructuring performed by O.

Definition 4.10. (Input provenance $\mathcal{I}$ w.r.t. result data item provenance ρ in result R of O) The input provenance $\mathcal{I}$ is a bag of triples $\langle i, I_j, \mathcal{A}\rangle$, where i is a data item from one of the input datasets of O, i,e., $i \in I_j, I_j \in I$, and $\mathcal{A}$ a set of paths recording the elements of i that are accessed by O to produce the result data item $r \in \rho$.

The above provenance model does not only contain information on the relationship between input and result data items of an operator (which is the previously mentioned lineage), it also records accesses and manipulations in $\mathcal{M}$ and $\mathcal{A}$ while transforming input items to result data items.

Example 4.11. To illustrate our model for structural provenance, we focus on the $flatten$ operator labeled 5 in Fig. 1. It unnests the nested items of attribute *user_mentions*. An excerpt from its input and output data is given at the top of Fig. 3. Black headers encode bag data types, light gray headers identify complex data items as nested data type. At the bottom left, Fig. 3 shows the provenance for the result items 42 and 43. The flatten operator derives item 42 from input item 1. It accesses path *user_mentions*[1] as recorded in $\mathcal{A}$. Further, it copies the first user of the *user_mentions* list (and implicitly, all paths in the path set $PS^{user\_mentions[1]}$) to the new item *m_user* as recorded in the mapping $\mathcal{M}$. Ignore the bottom right for now.

5 PROVENANCE CAPTURE

Based on the provenance model, we introduce inference rules describing the provenance capture of our supported operators. When the rules in Tab. 5 are annotated with a $*$, we extend an existing inference rule from [11]. Otherwise, we formalize the inference rule with and without provenance extension. Due to space constraints, we only show the complete set of inference rules with provenance. After explaining the map, flatten and aggregation rule, we show how we capture the structural provenance obtained from these rules efficiently.

5.0.1 Map. For the *map* operator, we assume that the function $\lambda(i)$ over input data item i returns a result of type data item, denoted as $\tau(\lambda(i)) \rightarrow \langle\ldots\rangle$. Given this precondition, without provenance capture, *map* returns the result of applying $\lambda(i)$, for each i in the input dataset I_1. The inference rule for the *map* operator in Tab. 5 additionally produces the provenance for each data item i. More formally, *map*, parameterized by a function λ, produces the result provenance $\mathcal{R}$, which is a bag of data items. Each data item extends the "normal" result of *map*, i.e., $\lambda(i)$ with two additional attributes: the input provenance $\mathcal{I}$ and mapping $\mathcal{M}$. For $\mathcal{I}$, the only input data item participating in producing an output data item $\lambda(i)$ is i, which originates from the single input dataset I_1. Because we generally do not know the internals of an arbitrary function λ, the set $\mathcal{A}$ is set to undefined, denoted by $\perp$. Thus, $\mathcal{I} = \{\{\langle i, I_1, \perp\rangle\}\}$. The structural mapping $M = \perp$ is also undefined because we have no knowledge of how elements from the input are restructured in the result.

Based on the rule for the *map* operator, we derive more general observations concerning our inference rules. The rules only capture structural provenance when the operator semantics clearly pinpoint paths to populate $\mathcal{A}$ and $\mathcal{M}$. Thus, the rule for the map operator captures the "undefined" semantics in $\mathcal{M} = \perp$ and $\mathcal{A} = \perp$. This semantics distinguishes from $\mathcal{M} = \emptyset$ and $\mathcal{A} = \emptyset$ semantics in the $filter$ and *union* rules. Both rules feature $\mathcal{M} = \emptyset$ since they do not restructure the data items. Each item's input structure is maintained in its entirety in the result. Further, the rule for the *union* operator holds $\mathcal{A} = \emptyset$ since it only performs an item-independent schema comparison of the input datasets.

5.0.2 Flatten. We introduced the flatten operator in Ex. 4.11 to illustrate our provenance model. Here, we explain the inference rule in Tab. 5 that captures the provenance for the $flatten$.

As preconditions, the rule requires the type of a_{col} to be either a list with duplicates (bag) or without duplicates (set). The result of the $flatten$ consists of items $r = \langle i, a_{new} : j\rangle$, where i refers

Filter*	$\dfrac{\varphi(i) \Rightarrow \text{true}}{I_1.filter[\varphi(i \in I_1)] \Rightarrow \{\{\langle i, \{\{\langle i, I_1, \cup p^i \in \varphi(i)\rangle\}\}, \emptyset\rangle \mid i \in I_1\}\}}$
Select*	$\dfrac{a_1, ..., a_n \in schema(I_1)}{I_1.select(a_1, ..., a_n) \Rightarrow \left\{\left\{\left\langle r, \left\{\left\{\left\langle i, I_1, \bigcup_{k=1}^{n} (a_k)^i \right\rangle\right\}\right\}, \bigcup_{k=1}^{n} \langle (a_k)^i, (a_k)^r \rangle \right\rangle \mid r = \langle i.a_1, ..., i.a_n\rangle, i \in I_1\right\}\right\}}$
Map*	$\dfrac{\tau(\lambda(i)) \Rightarrow \langle ... \rangle}{I_1.map[\lambda(i \in I_1)] \Rightarrow \{\{\langle \lambda(i), \{\{\langle i, I_1, \perp\rangle\}\}, \perp\rangle \mid i \in I_1\}\}}$
Join	$\dfrac{\varphi(i, j) \Rightarrow \text{true}}{I_1.join[\varphi(i \in I_1, j \in I_2)](I_2) \Rightarrow \left\{\left\{\left\langle r, \left\{\left\{\left\langle i, I_1, \bigcup_{p^i \in \varphi(i,j)} p^i \right\rangle, \left\langle j, I_2, \bigcup_{q^j \in \varphi(i,j)} q^j \right\rangle\right\}\right\}, \left\{\langle p^i, p^r\rangle \mid p^i \in schema(I_1)\right\} \cup \left\{\langle q^j, q^r\rangle \mid q^j \in schema(I_2)\right\}\right\rangle \mid r = \langle i, j\rangle, i \in I_1, j \in I_2\right\}\right\}}$
Union*	$\dfrac{\tau(I_1) = \tau(I_2)}{I_1.union(I_2) \Rightarrow \{\{\langle i, \{\{\langle i, I_1, \emptyset\rangle\}\}, \emptyset\rangle \mid i \in I_1\}\} \uplus \{\{\langle j, \{\{\langle j, I_2, \emptyset\rangle\}\}, \emptyset\rangle \mid j \in I_2\}\}}$
Flatten	$\dfrac{\tau(a_{col}) \Rightarrow \{\{\}\} \vee \tau(a_{col}) \Rightarrow \{\}}{I_1.flatten(a_{new}, explode(a_{col})) \Rightarrow \{\{\langle r, \{\{\langle i, I_1, \{\langle (a_{col}[x])^i\rangle\}\rangle\}\}, \{\langle (a_{col}[x])^i, a^r_{new}\rangle\}\rangle \mid r = \langle i, a_{new} : j\rangle, i \in I_1, j \in i.a_{col} \text{ at position } x\}\}}$
Grouping*	$\dfrac{G = \{\{g_1, ..., g_n\}\} \qquad \pi_G(i) = \langle g_1 : i.g_1, ..., g_n : i.g_n\rangle}{I_1.groupBy(g_1, ..., g_n) \Rightarrow \{\{\{\{\langle i, \{\{\langle i, I_1, \{g_1, \ldots, g_n\}\rangle\}\}, \emptyset\rangle \mid i \in I_1, \pi_G(i) == j\}\} \mid j \in set(\{\{\pi_G(i) \mid i \in I_1\}\})\}\}}$
Aggregation	$\dfrac{\begin{array}{c} \tau(I) == \{\{\tau(I_1), ..., \tau(I_n)\}\} \qquad \tau(I_1) == \ldots == \tau(I_n) == \{\{...\}\} \\ A_c == \left\{\alpha_{c_1}(a_1), ..., \alpha_{c_m}(a_m)\right\} \text{ with } \tau(\alpha_{c_k}(a_k)) \Rightarrow c \qquad A_B == \left\{\alpha_{B_1}(b_1), ..., \alpha_{B_p}(b_p)\right\} \text{ with } \tau(\alpha_{B_k}(b_k)) \Rightarrow \{\{...\}\} \\ \forall I_k \in I, \forall i, j \in I_k, \pi_G(i) == \pi_G(j) \text{ with } G == schema(I_k) \setminus \{a_1, \ldots, a_m, b_1, \ldots, b_p\} \end{array}}{I.agg(A_c, A_B) \Rightarrow \left\{\left\{\left\langle \begin{array}{l} r = \left\langle d, a_{\alpha_{c_1}} : \alpha_{c_1}\left(\pi_{a_1}(I_k)\right), \ldots, a_{\alpha_{c_m}} : \alpha_{c_m}\left(\pi_{a_m}(I_k)\right), a_{\alpha_{B_1}} : \alpha_{B_1}\left(\pi_{b_1}(I_k)\right), \ldots, a_{\alpha_{B_p}} : \alpha_{B_p}\left(\pi_{b_p}(I_k)\right)\right\rangle, d \in set(\{\{\pi_G(i) \mid i \in I_k\}\}), I_k \in I, \\ \left\{\left\{\left\langle i, I_k, \bigcup_{g \in G} g^i \cup \bigcup_{a \in A_c} a^i \cup \bigcup_{b \in A_B} b^i \right\rangle \mid i \in I_k, I_k \in I\right\}\right\}, \\ \left\{\langle g^i, g^r\rangle \mid g \in G, i \in I_k\right\} \cup \left\{\left\langle a^i_k, \left(a_{\alpha_{c_k}}\right)^r\right\rangle \mid k = 1, ..., m, i \in I_k\right\} \cup \left\{\left\langle a^i_k, \left(a_{\alpha_{B_k}}\right)^r\right\rangle \mid k = 1, ..., p, i \in I_k\right\} \end{array} \right\rangle\right\}\right\}}$

Table 5: Provenance capture semantics partially based on operator semantics from [11]. Access $\mathcal{A}$ and manipulation $\mathcal{M}$ provenance is highlighted.

to the input item and a_{new} to the newly created attribute. This attribute holds item j that is unnested from i's attribute a_{col}, i.e., $i \in I_1, j \in i.a_{col}$. In our structural provenance, we need to refer to the position of j within its bag (or set) in the context of i. Therefore, we denote the position of j in $i.a_{col}$ by pos. For each result item r, the structural provenance is $\rho = \langle r, \mathcal{I}, \mathcal{M}\rangle$ with $\mathcal{I} = \left\{\left\{\left\langle i, I_1, \left\{(a_{col}[pos])^i\right\}\right\rangle\right\}\right\}$ and $\mathcal{M} = \left\{\left\langle (a_{col}[pos])^i, a^r_{new}\right\rangle\right\}$. Here, $(a_{col}[pos])^i$ denotes the access path on the pos-th element of attribute a_{col} in the context of the input item i. a^r_{new} is the path to the new attribute in the context of the result item r.

5.0.3 Aggregation. The *aggregation* in Tab. 5 requires a bag of equally structured input collections (we only show bags for conciseness) as input, i.e., $\tau(I) = \{\{\tau(I_1), \ldots, \tau(I_n)\}\}$ such that $\tau(I_1) == \ldots == \tau(I_n) == \{\{\ldots\}\}$. These nested bags are constructed by the *grouping*. Further, the *aggregation* supports multiple aggregation functions. Among those supported by data analytics systems, we distinguish between aggregation functions that, given a bag as input, return an atomic constant value c (e.g., *count*, *sum*, *max*) and aggregation functions returning nested collections (e.g., *collect_list* and *collect_set*). We denote these by A_c and A_B, respectively. The rule also requires that all attributes G that are not aggregated by either a function in A_c or A_B, but that are present in the schema of a collection $I_k \in I$ are equal.

Given these preconditions, *aggregation* reduces each of the nested bags $I_k \in I$ to a single data item. It returns the unique value present in I_k for non-aggregated attributes in G and the results of the specified aggregate functions that are applied on the input attributes. The result of this process is the item r in the first line at the bottom of the *aggregation* rule. The second line shows $\mathcal{I}$. A result item r is based on input items that all originate from the same input collection $I_k \in I$. Thus, for each $i \in I_k$, the rule creates a data item $\langle i, I, \mathcal{A}\rangle \in \mathcal{I}$. The set of attribute accesses performed during aggregation includes the paths to all attributes in G, the paths to all attributes aggregated by functions in A_c, and the paths to all attributes aggregated by functions in A_B. That is, $\mathcal{A} = \bigcup_{g \in G} g^i \cup \bigcup_{a \in A_c} a^i \cup \bigcup_{b \in A_B} b^i$. For $\mathcal{M}$ in line 3, the rule maps aggregated attributes to the newly created attributes of r, which hold the aggregated items.

5.1 Lightweight provenance capture

The provenance capture rules have the potential for optimization, since they hold redundant information. First, recording a unique identifier suffices to identify each top-level item. Second, recording the paths accessed and manipulated on a schema level once per operator suffices since the paths are the same for all processed data items. They only differ in the identifier of the top-level item and the positions of items in nested collections. The lightweight operator provenance $\mathcal{P}$ exploits these observations to keep overhead at capture time low.

Definition 5.1. (Operator provenance $\mathcal{P}$) The operator provenance $\mathcal{P}$ is the following 5-tuple:

$$\mathcal{P} = \langle oid, type, \mathcal{I} : \{\{\langle p, \mathcal{A}\rangle\}\}, \mathcal{M}, P\rangle$$

$\mathcal{P}$ has an operator identifier *oid* and a *type*. The bag $\mathcal{I}$ holds one tuple for each of the operator's inputs. This tuple holds a reference to the preceding operator p and the paths accessed $\mathcal{A}$ on the input at a schema level. They are data item independent. Similarly, $\mathcal{P}$ has a bag of manipulated paths $\mathcal{M}$ on a schema level. Positions of items in nested bags are replaced with placeholders. The bag P in $\mathcal{P}$ holds the unique identifiers of the top-level input

Operator	Provenance structure
$map, select, filter$	$P = \{\{\langle id^i, id^o \rangle\}\}$
$join, union$	$P = \{\{\langle id^i_1, id^i_2, id^o \rangle\}\}$
$flatten$	$P = \{\{\langle id^i, pos, id^o \rangle\}\}$
$groupby$ and $aggregation$	$P = \{\{\langle ids^i : \{\{id^i\}\}, id^o \rangle\}\}$

Table 6: Operator-dependent provenance structure

and output items, as well as positions of accessed or manipulated items in nested collections, if needed.

The content of P depends on the operator type as summarized in Tab. 6. In this table, the attributes id^i and id^o hold the unique identifiers of top-level items in the input and the output, respectively. If the operators have multiple inputs, attributes id^i_1 and id^i_2 are indexed in the order of appearance in $\mathcal{I}$. The structure P of the flatten operator has a reference to the *pos*ition of the nested item being flattened. The aggregation holds a collection of input ids^i for each group. The position of the input id^i is equal to the position of any nested item that the aggregation produces.

Example 5.2. Fig. 3 shows the reduced operator provenance $\mathcal{P}_5$ for the flatten operator at the bottom right.

In the following section, we describe how the backtracing algorithm computes the structural provenance of nested data from the lightweight provenance structures $\mathcal{P}$.

6 BACKTRACING

Querying the provenance of items or structures in the result involves two major phases. In the first phase, the backtracing algorithm identifies those data items, for which a user queries provenance (Sec. 6.1). In the second phase, it traces these items back to the input data (Sec. 6.3).

6.1 Structural query processing

DISC systems have rudimentary means to address individual nested items, at most. They lack sophisticated means to address arbitrary combinations of them, which is essential for querying structural provenance. Thus, we devise an extension to a DISC system (i.e., Apache Spark) to support tree-pattern queries. Tree-patterns allow for addressing combinations of nested items that are related by their structure [13]. Intuitively, they express structural queries in the form of a tree, in which each node represents an attribute and edges define parent-child or ancestor-descendant relationships (depending on edge type) that should exist between two connected nodes. Further constraints may be imposed on attribute nodes, e.g., equality of an attribute's value to a constant. Therefore, we define a novel distributed tree-pattern matching algorithm to return the query result in an efficient and scalable way. Due to space constraints, we omit details on processing tree-pattern queries but show an example.

Example 6.1. Fig. 4 shows a tree-pattern for the provenance question introduced in Sec. 2. Its *root* has an ancestor-descendant edge to the *id_str* node. All other edges indicate parent-child relationships. The *id_str* and *text* nodes hold equality conditions, which require values of those attributes to be equal to *lp* and *Hello World*, respectively. Further, as indicated by the black box, the value *Hello World* has to occur twice in the nested collection.

Our algorithm matches the tree-pattern against a dataset D (in our example, the final result of the processing pipeline) to then return the matching data in the form of a backtracing structure, which we introduce next.

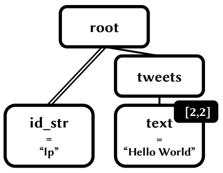


Figure 4: Example tree-pattern in a provenance question

6.2 Backtracing structure

The backtracing structure describes the items that are queried in the provenance question and traced back to the input. The backtracing algorithm updates its content while stepping backward.

Definition 6.2. (Backtracing structure) The backtracing structure $\mathcal{B} = \{\{\langle id, \mathcal{T} \rangle\}\}$ is a bag of provenance identifiers id of top-level data items associated with a backtracing tree $\mathcal{T}$ (see next definition), referencing attributes in the schema of id.

The nodes in the backtracing trees also hold information about the access and manipulation of the attribute and whether the attribute is contributing or just influencing the items queried.

Definition 6.3. (Backtracing tree) The backtracing tree $\mathcal{T} = \langle root, N \rangle$ holds complex nodes $n \in N$. Each node $n = \langle name, parent, C, A, M, c \rangle$ has a $name$ equal to the attribute name it references. Further, it references its parent node p, and its children C. A node also holds the set of operators A that access the referenced attribute and a set of operators M that manipulate the attribute. A boolean value c indicates whether the attribute contributes to the items in the provenance question ($c = true$) or whether it influences the items ($c = false$).

Example 6.4. Examples of backtracing trees are provided in Fig. 2. The right tree corresponds to the backtracing tree obtained from matching the tree-pattern in Ex. 6.1 on the data in Tab. 2. The left trees correspond to backtracing trees resulting from recursively updating the backtracing structure while stepping back through the processing pipeline to the input data.

The following two methods manipulate the trees, and the backtracing algorithm calls them repeatedly during backtracing. Their execution context is an instance of an operator provenance $\mathcal{P}$ and a backtracing structure $\mathcal{B}$.

The $manipulatePath$ method performs two tasks. First, it manipulates the nodes in $\mathcal{T}$. For each input and output path in $m \in \mathcal{P}.M$ it transforms the output path back to the specified input path in m, if the output path exists in $\mathcal{T}$. After the transformations, the nodes in the tree $\mathcal{T}$ conform to the schema of the input. Second, it adds the current operator identifier $\mathcal{P}.oid$ to each node's manipulation collection M.

The $accessPath$ method records access to attributes in the nodes of $\mathcal{T}$. During that process, one of two cases applies. In the first case, all nodes of the path $a \in \mathcal{A}, \mathcal{A} \in \mathcal{P}.\mathcal{I}$ already exist in $\mathcal{T}$. Then the method adds the $\mathcal{P}.oid$ to each node's access collection A. In the second case, nodes in path a do not exist in $\mathcal{T}$, because these attributes are neither needed to reproduce the result nor have been accessed by other operators so far. Then, the $accessPath$ method adds the according nodes to $\mathcal{T}$ but sets the contribution value to $c = false$ since these nodes are not required to reproduce the queried data items.

6.3 Backtracing algorithm

Alg. 1 shows the backtracing algorithm that traces the queried items recursively back from the result to the input. It takes the

Algorithm 1: backtrace($\mathcal{P}, \mathcal{B}$)

```
    Input: $\mathcal{P}$, $\mathcal{B}$
    Output: $\mathcal{B}$
1   switch $\mathcal{P}.type$ do
2       case “$filter$” do
3           $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow$ backtraceFilter($\mathcal{P}$, $\mathcal{B}$)
4       case “$flatten$” do
5           $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow$ backtraceFlatten($\mathcal{P}$, $\mathcal{B}$)
6       ...
7   if $\mathcal{P}'$ is defined then
8       backtrace($\langle \mathcal{P}', \mathcal{B}' \rangle$)
9   return $\mathcal{B}'$
```

Algorithm 2: backtraceFlatten($\mathcal{P}, \mathcal{B}$)

```
    Input: $\mathcal{P}$, $\mathcal{B}$
    Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1   $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow backtraceOperatorGeneric(\mathcal{P}, \mathcal{B})$
2   $\mathcal{B}' \leftarrow \alpha_{mergeTrees(\mathcal{T},pos)}(\gamma_{id}(\mathcal{B}'))$
3   return $\langle \mathcal{P}', \mathcal{B}' \rangle$
```

Algorithm 3: backtraceOperatorGeneric($\mathcal{P}, \mathcal{B}$)

```
    Input: $\mathcal{P}$, $\mathcal{B}$
    Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1   $\mathcal{B}' \leftarrow \pi_{id^i \rightarrow id, pos, \mathcal{T}}(\mathcal{P}.P \bowtie_{id^o = id} \mathcal{B})$
2   for $t \in \mathcal{B}'.\mathcal{T}$ do
3       for $m \in \mathcal{P}.\mathcal{M}$ do
4           $manipulatePath(t, m, \mathcal{P}.oid)$
5       for $a \in \mathcal{P}.\mathcal{I}_1.\mathcal{A}$ do
6           $accessPath(t, a, \mathcal{P}.oid)$
7   return $\langle \mathcal{P}' \leftarrow \mathcal{P}.\mathcal{I}_1.p, \mathcal{B}' \rangle$
```

operator provenance of the last operator $\mathcal{P}$ in the pipeline and the backtracing structure $\mathcal{B}$ as input to call the operator-dependent backtracing method, which returns its predecessor's operator provenance $\mathcal{P}'$ and an updated backtracing structure $\mathcal{B}'$. The algorithm recursively calls the backtrace method until the input is reached. Then $\mathcal{P}'$ is not defined so that the recursion ends.

Flatten, Select, Filter, Map. As shown in Alg. 2, backtracing the flatten operator has two steps. In the first step (l. 1), the algorithm calls *backtraceOperatorGeneric* (Alg. 3) to undo the flatten on each item in $\mathcal{B}$ individually. At this step, the algorithm does not consider positions in the flattened collection. As a result, it obtains $\mathcal{P}'$ and $\mathcal{B}'$. In the second step (l. 2), the algorithm groups the trees and positions in $\mathcal{B}'$ by the top-level item *id* and merges all trees of the same *id*, considering the position *pos*.

The generic backtracing algorithm, shown in Alg. 3, also has two major steps. In the first step (l. 1), it joins B with the provenance associations P of $\mathcal{P}$ to obtain the input identifiers of the top-level items along with the trees in B (l. 1). These identifiers become the new *ids* in $\mathcal{B}'$ so that they match the id^o of the projection's predecessor $\mathcal{P}.\mathcal{I}_1.p, \mathcal{B}'$. This join is essentially the same one that existing lineage solutions [15, 17, 22] apply for backtracing. In the second step (ll. 2-6), the algorithm iterates over all the trees in $\mathcal{B}'$ to undo all recorded structural manipulations in the *manipulatePath* method and record the access to attributes in the *accessPath* method.

Example 6.5. The example input of Alg. 2 is $\mathcal{P}_5$ in Fig. 3 (bottom right) and a backtracing structure $\mathcal{B}$ with the two items of $id = 42$ and $id = 43$. They reference the items with the same identifier in Fig. 3. For simplicity, the example subtree $\mathcal{T}$ is reduced to the path *m_user.id_str*. The *backtraceFlatten* algorithm calls the *backtraceOperatorGeneric* algorithm (Alg. 2, l. 1), which joins P_5 with $\mathcal{B}$ (Alg. 3, l.1). Afterwards, both items are assigned $id = 1$. Then the algorithm modifies the trees to $user\_mentions.[pos].id_{str}$ so that they comply with the input schema of the flatten operator (Alg. 3, l.4). However, instead of holding the position of the nested items, they hold [*pos*] placeholders. The *mergeTrees* method in Alg. 2 (l. 2) replaces them with $pos = 1$ and $pos = 2$ for the items with the former $id = 42$ and $id = 43$, respectively. Further, it merges their trees because both items have $id = 1$ and, thus, are grouped together. Finally, the algorithm returns a $\mathcal{B}'$ with the item of $id = 1$ and the tree referring to positions 1 and 2 in the nested collection *user_mentions*.

Algorithm 4: backtraceAggregation($\mathcal{P}, \mathcal{B}$)

```
    Input: $\mathcal{P}$, $\mathcal{B}$
    Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1   $P^* \leftarrow pos\_flatten(\mathcal{P}.P.ids^i, id^i, p_P)$
2   $\mathcal{B}' \leftarrow P^* \bowtie_{id^o = id} \mathcal{B}$
3   $\mathcal{B}' \leftarrow withCol(\mathcal{B}', inProv, false)$
4   for $b \in \mathcal{B}'$ do
5       for $m \in \mathcal{P}.\mathcal{M}$ do
6           if $contains(m.out, [pos])$ then
7               $out \leftarrow replace(m.out, b.p_P)$
8           else
9               $out \leftarrow m.out$
10          if $out \in b.\mathcal{T}$ then
11              $b.inProv = true$
12              $manipulatePath(b.\mathcal{T}, \langle m.in, out \rangle, \mathcal{P}.oid)$
13          $removeNodes(b.\mathcal{T}, m.out)$
14  for $t \in \mathcal{B}'.\mathcal{T}$ do
15      for $a \in \mathcal{I}.A$ do
16          $accessPath(t, a, \mathcal{P}.oid)$
17  $\mathcal{B}' \leftarrow \pi_{id^i \rightarrow id, \mathcal{T}}(\sigma_{inProv=true}(\mathcal{B}'))$
18  return $\langle \mathcal{P}' \leftarrow \mathcal{I}.p, \mathcal{B}' \rangle$
```

The algorithms to backtrace a select, filter, or map operator are basically the same as Alg. 3, except that they do not project on the *pos* attribute (l. 1). Some optimizations are applicable to the filter. Since the filter does not manipulate any data, its backtracing algorithm does not loop over the manipulations $\mathcal{P}.\mathcal{M}$. The backtracing algorithm for the map operator has no information on the paths manipulated or accessed. Thus, it marks all nodes in the input schema as manipulated by default.

Aggregation and Nesting. As described in Sec. 5, aggregation and nesting are preceded by a grouping. Further, our model allows multiple aggregations and nestings over different attributes. Alg. 4 describes the procedure to trace aggregation and nesting back to the input of the preceding grouping.

Unlike the provenance structures of other operators, $\mathcal{P}.P$ of an aggregation holds a nested collection of input ids^i (cf. Tab. 5.1). Thus, Alg. 4 first flattens the ids^i and their positions into the columns id^i and p_P, respectively (l. 1). After joining P^* with $\mathcal{B}$ to $\mathcal{B}'$ (l.2), the algorithm adds a column *inProv* to $\mathcal{B}'$ (l. 3). This column is initialized with $false$ and used later to indicate, whether items in $\mathcal{B}'$ remain in the backtraced provenance. Then, the algorithm iterates over each item in $\mathcal{B}'$ and each manipulated path in $\mathcal{P}.\mathcal{M}$ (ll. 4-13). For each manipulated path $m.out$, it checks for a position placeholder [*pos*] (l.6), which only occurs when the operator performs bag nesting. In this case, the input item with id^i contributes exactly to the item in the nested bag that also has position p_P. Thus, the algorithm replaces the placeholder in $m.out$ and stores the result in *out* (l. 7). Otherwise, *out* is assigned $m.out$ (l. 9). If the exact path *out* is in the provenance tree, item *b* is marked as relevant and the path is adjusted accordingly (ll.10-12). In case of a bag nesting, the provenance tree may also hold information of items at other positions. The algorithm removes these nodes calling the *removeNodes* method (l. 13). It marks the accessed paths, to which the grouping attributes also belong (ll.14-16). In a final step, the algorithm removes all items and attributes from $\mathcal{B}'$ that are irrelevant for further backtracing. Their value in *inProv* was not set to true.

Example 6.6. Let us apply Alg. 4 on the nesting operator in our running example, which collects all tweeted texts in a nested bag. The backtracing structure $\mathcal{B}$ contains just the item with $id = 102$ and the right tree $\mathcal{T}$ of Fig. 2, which refers to the duplicate text *Hello World*. The operator Provenance $\mathcal{P}$ contains a provenance structure P that Alg. 4 flattens out to the following P^*:

id^i	p_P	id^o
81	1	102
82	2	102
93	3	102
95	4	102

After joining $\mathcal{B}$ with P^*, each entry with $id^o = 102$ holds a copy of the same tree $\mathcal{T}$. For a single loop iteration over $\mathcal{B}'$ and $\mathcal{M}'$ (ll. 4-13), we choose $b \in \mathcal{B}'$ with $id^i = 82$, $p_P = 2$, $id^o = 102$ and path $m.out = tweets.[pos]$. The algorithm replaces placeholder $[pos]$ with 2, so that $out = tweets.2$ (l. 7). Since out is part of $\mathcal{T}$, it sets $b.inProv = true$ (l. 11) and transforms the subtree $tweets.2.text$ to the subtree $text$ (l. 12). Then, it removes the node $tweets$ and all its children from its copy of $\mathcal{T}$ (l. 13), which includes the nodes in path $tweets.3.text$. Now, the nodes in $\mathcal{T}$ describe a subset of the schema of the aggregation's input data. The algorithm marks the accessed attributes and removes all items that are not part of the provenance. Here, it marks the $user$ and its children as accessed (ll. 14-16), since these attributes are used for grouping. Further, it removes items from $\mathcal{B}'$ whose provenance is not queried (l. 17), e.g., b' with $id^i = 95$ and $p_P = 4$.

Join and Union. Unlike the other operators, the join and the union operator have two predecessors. Thus, the backtracing algorithms require an additional parameter to specify which of the two inputs is traced back to. Based on that parameter, the algorithms pick the appropriate input tuple from the operator provenance $\mathcal{P}.\mathcal{I}$ and the appropriate input identifiers id_1^i or id_2^i from the provenance structure $\mathcal{P}.P$ (cf. Tab. 4). Then they call the generic backtracing algorithm from Alg. 3. Afterwards, the algorithm for the join operator removes all nodes in the provenance trees $\mathcal{B}'.\mathcal{T}$ that are not part of the chosen input schema, since they reference elements in the schema of the other input. The algorithm for the union operator filters out all items in $\mathcal{B}'$ whose value is undefined in the chosen field id_1^i or id_2^i . These items originate from the other input of the union operator.

7 IMPLEMENTATION & EVALUATION

We integrate the contributions described in this paper into a system prototype named Pebble. Sec. 7.1 provides some details of the system implementation, demonstrated in [9]. Sec. 7.2 then describes our evaluation setup and workload, which we use for our experimental evaluation (Sec. 7.3).

7.1 Implementation

While our contributions are generally applicable to DISC systems, we implement Pebble as a library extension for Apache Spark. This allows us to better compare it to Titian, which is the only other fully integrated provenance solution for DISC systems that has been implemented over Spark [16].

Fig. 5 shows Pebble's architecture (blue) on top of the *SparkSQL* API (grey), which is independent of the *Spark Core* module and further modules (grey) such as the MLlib. To provide a transparent user experience, Pebble has an API wrapper *PebbleAPI*. It directs user requests to the *SparkSQL* module or the *Pebble Core* module, which contains the *Capture* and *Query* submodules.

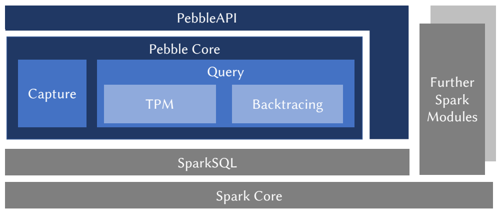


Figure 5: Pebble's architecture

S	Description (detailed descriptions available in [8])
T1	filters tweets containing the text *good*, flattens and groups by the mentioned users to collect a bag of complex tweet objects
T2	flattens the nested lists *hashtags*, *media*, *user mentions*
T3	running example
T4	associates all occurring hashtags with the authoring and mentioned users
T5	finds all users that tweet about *BTS*, and are mentioned in a *BTS* tweet
D1	associates inproceedings from 2015 with the their according proceeding(s)
D2	unites and restructures conference proceedings and articles
D3	computes nested list for aliase, co-authors, and works per author
D4	computes nested list of all associated inproceedings for each proceeding
D5	is D4 extended with a UDF in map that returns the number of authors per proceeding

Table 7: Short informal scenario descriptions

The former submodule extends Spark's dataframes and operators to capture the structural provenance as described in Sec. 5. The latter submodule implements the backtracing algorithm from Sec. 6. It utilizes maps to represent the provenance trees $\mathcal{T}$ and modifies the tree in place with user-defined functions. Each of the Algs. 2-4 iterate over all items in the backtracing structure $\mathcal{B}$ and perform changes impacting only one item at a time. Thus, the for-loops with iterator variables t or b in Algs. 2-4 are parallelized across the DISC system. However, the backtracing needs to be called for each input dataframe independently, because Spark operators always generate just one result dataframe.

7.2 Test setup & workload

For our experimental validation, we run Pebble on a cluster with three worker nodes, each having 8 cores, 256GB main memory, and SSD storage. All nodes run Scala 2.11, Hadoop 3.1.0 and Spark 2.3.1. We average five test runs framed in an additional warm-up and cool-down run. The error bars displayed in our graphs show the standard deviation. We write the result to disk to ensure that Spark computes the full result. Otherwise, Spark "optimizes" attributes away. The experiments run on 100GB input data, if not mentioned otherwise.

We base our evaluation on a nested Twitter and a DBLP dataset. We scale the datasets from 100GB to 500GB in steps of 100GB. For each of the datasets, we define five scenarios containing a Spark program to be executed with and without provenance capture and a corresponding structural query. Each supported operator occurs at least once in the scenarios. The Twitter dataset contains up to 130 million tweets (500GB). Each tweet has up to 1000 attributes and eight layers of nesting [27]. We define five test scenarios T1 - T5 (Tab. 7). The DBLP dataset contains up to 1.5 billion records (500GB) that are extracted from the dblp.xml. Records have one of ten types such as article or proceeding [21]. They are split by type and upscaled, such that important characteristics such as the average number of inproceedings per proceeding are preserved. We define five test scenarios D1 - D5 (Tab. 7).

7.3 Experimental evaluation

We conduct experiments to study the runtime and space overhead when capturing lightweight structural provenance. We also evaluate the performance of querying the structural provenance. Further, we perform a comparative evaluation with Titian [17],

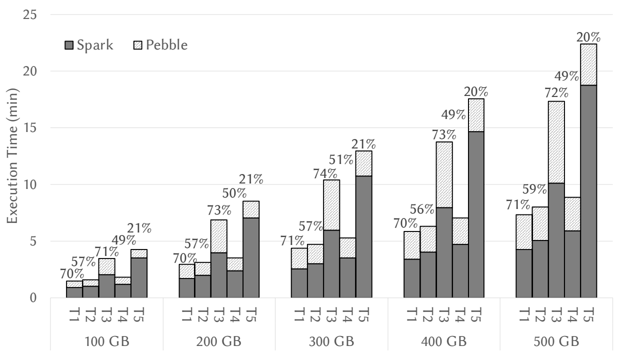


Figure 6: Runtime overhead on Twitter dataset

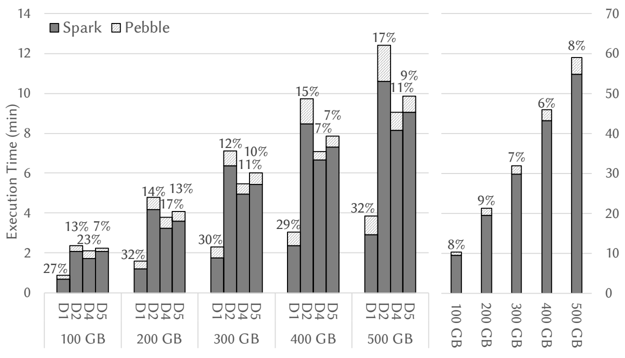


Figure 7: Runtime overhead on DBLP dataset (right: D3)

kindly provided to us by the authors, and fully lazy provenance capture as described by PROVision [28]. We conclude our evaluation with a use-case analysis for auditing and data-usage patterns.

7.3.1 Capture runtime overhead. The first series of measurements shows the runtime overhead imposed by the lightweight provenance capture for increasing data sizes. Our goal is to show (i) how Pebble scales over the input data size and (ii) how data size affects the runtime overhead.

We measure the execution time for each of the Twitter scenarios T1 to T5, once without provenance capture, i.e., with Spark's regular operator semantics, and once with Pebble's provenance capture as defined in Sec. 5.1. Fig. 6 shows the results on datasets from 100GB to 500GB. The solid dark grey part of each part shows the runtime that Spark requires without provenance collection. The textured light grey part on top of each bar shows the overhead when running provenance capture. The percentage on top of the textured bars indicate the relative overhead between the former and the latter types of experiments. Analogously, Fig. 7 reports runtimes for scenarios D1 to D5.

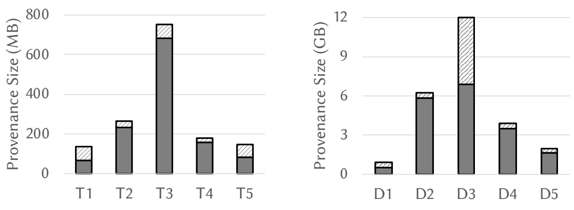
As expected, across all experiments, the runtime increases when provenance is collected since Pebble performs extra work. Runtime with and without provenance grows linearly with the data size. As the overhead percentages indicate, the relative overhead imposed by provenance capture remains constant with increasing data sizes for most scenarios. Thus, we conclude that Pebble scales with the input data size. However, the relative overhead varies significantly between the scenarios. It ranges from 75% (T3) down to 8% (D3, shown on the right of Fig. 7). A detailed analysis of D3 reveals that spilling large final and intermediate results to disk – or more generally speaking disk I/O – dominates the runtime. The time to compute the extra provenance is small. In contrast, scenario T3 reads the input tweets twice to perform a union operation. As a consequence, Pebble annotates the input data twice during provenance capture, hindering Spark to optimize reading the input.

We further investigate overhead incurred by Pebble for each individual operator (no graphs shown due to space constraints). Overall, the overhead highly depends on the size ratio between the collected provenance and the processed input data. In general, for operators with constant provenance annotation overhead (filter, select, union, join, and flatten), the relative overhead decreases with an increasing number of attributes in the input data. In the case of the DBLP dataset, which has less than 50 attributes, the overhead ranges between 5% and 25% for the mentioned operators. The overhead is particularly high for aggregations that reduce many input items to a single value. Then, Pebble stores a collection with all item identifiers contributing to the aggregated item. This collection typically is orders of magnitude larger than the result item itself. Consequently, the observed runtime overhead exceeds 100% of the actual execution time for the aggregation. However, even this overhead is negligible when disk I/O operations dominate the execution time.

7.3.2 Capture space overhead. The second series of measurements shows the space required to store captured provenance. We show (i) that the provenance size depends on dataset and scenario characteristics, (ii) that large provenance sizes do not necessarily correlate with high runtime overhead, and (iii) that the captured structural provenance typically adds an overhead of less than 200MB compared to lineage. Thus, it typically does not significantly affect scalability. Results are reported in Fig. 8(a) and Fig. 8(b). The dark grey part of each bar shows the size of lineage for top-level items and the stacked and textured bars show the additional space required by structural provenance.

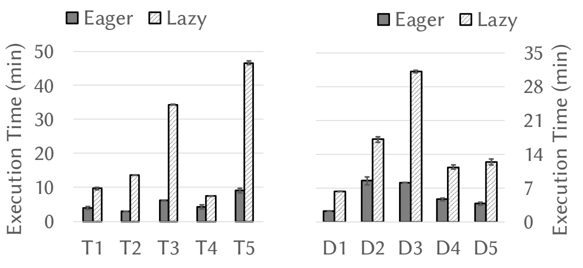
The y-axis of the Twitter graph has a Megabyte scale, whereas the y-axis of the DBLP graph has a Gigabyte scale. The reason is that the items in the Twitter dataset have about 1000 attributes, whereas the items in the DBLP dataset have less than 50 attributes. Therefore, 100GB of DBLP data contain more than 100 times as many data items as 100GB of Twitter data. Given that Pebble associates identifiers to top-level data items only, it stores more than 100 times the annotations for DBLP scenarios compared to the Twitter scenarios. Hence, the DBLP provenance is orders of magnitude larger than the Twitter provenance and lets us conclude that the size of the provenance significantly depends on the number of tracked top-level data items in the input.

Further, the sizes significantly differ among the scenarios of the same dataset. For instance, the provenance of scenario T3 amounts to 750MB, 5.5 times the size of T1's provenance. There are three reasons for the different size: (i) As mentioned above, in T3, our solution annotates the input data twice; (ii) The processing pipeline of T3 consists of 7 processing steps that trigger provenance collection, whereas the pipeline of T1 only consists of 5 steps; (iii) The filter in T1 reduces the total amount of tracked



(a) Twitter dataset (b) DBLP dataset

Figure 8: Size of collected structural provenance



(a) Twitter dataset (b) DBLP dataset

Figure 9: Runtime of Pebble's backtracing

data items early in the pipeline. Clearly, space overhead also depends on the number of operators in a program and the number of top-level items in intermediate results.

The scenarios T3 and T1 have comparable runtime overhead (see Fig. 6(a)) of around 70 - 75% and, thus, highest across all scenarios. While scenario T3 is also the scenario with the largest provenance size, T1 has a comparable runtime overhead but a much smaller provenance size. Similarly, the relationship of the runtime and space overhead is actually the inverse for scenario D3 and D1 in Fig. 8(b). D3 has the largest and D1 the smallest provenance size. However, D1 has a runtime overhead of 27%, whereas D3 only has 7%. Therefore, it is not generally true that a high runtime overhead correlates with a high space overhead.

Looking at the space overhead of lineage and structural provenance, we see that in most scenarios, structural provenance takes less than 200MB additional space, even in scenarios where lineage itself takes Gigabytes. The only exception is D3, where a flatten occurring early in the pipeline followed by a very selective join causes the comparatively high size difference.

Concerning the above experiments, which are all related to provenance capture, we make the following general observations. The provenance size highly depends on the number of top-level items in the input and intermediate results. As explained, the provenance size may not correlate with the runtime overhead. Other factors, such as processing optimizations, data width, or significant disk I/O potentially have a higher impact on the relative runtime overhead than the provenance size. While the size difference between lineage and structural provenance is small in many practical scenarios, the overhead can increase when flatten operators store positions that lineage solutions do not capture.

7.3.3 Querying structural provenance. Our third series of experiments focuses on processing structural queries over provenance-annotated result datasets. The runtimes reported in Fig. 9 include both the tree-pattern matching on the program's result items and the time needed to backtrace these result items to the input with the help of structural provenance. We report results on query processing time in our holistic approach (i.e., where structural provenance has been eagerly captured and is traced back). We also implement a fully lazy query approach that can be considered an extension of PROVision [28] to our processing pipelines. The query runtime for these two approaches, labeled eager and lazy respectively, are shown for both the Twitter scenarios (Fig. 9(a)) and the DBLP scenarios (Fig. 9(b)).

The graphs in Fig. 9 do not explicitly show the time for tree-pattern matching since the matching is integrated into Spark's processing pipeline. It becomes part of Spark's execution plan and undergoes optimizations such as filter push down. Therefore, time cannot be measured independently in a reliable way.

The dark bars in Fig. 9 show that querying structural provenance (eagerly) takes more time than the actual program execution (cf. Fig. 6 and 7). We identify two reasons for this behavior: (i) the backtracing presented in Sec. 6 performs a join operation for each operator in the actual program, even for computationally less expensive operators such as filters and selects. (ii) Backtracing has to manipulate the provenance trees for each operator.

When comparing the performance of our holistic capture and query approach with a completely lazy query approach such as PROVision [28], we see that our holistic provenance querying approach (eager) is always faster than the lazy approach. In the scenarios T3, T5, and D3 the difference amounts to a factor four to seven for two reasons. First, lazy processing needs to trace back result items for each input dataset independently and these scenarios have multiple input datasets. Hence, the extra time to query provenance lazily add up for each input. Second, the processing pipelines in these scenarios are deep, yielding high provenance query times for each input dataset.

Based on the above experiments, we draw the following conclusions for provenance querying. Lazily querying structural provenance is less attractive the more operators a program has and the more input datasets it processes. It is less time consuming to rerun a program with provenance capture and query the provenance eagerly than using lazy provenance querying approaches such as [28].

7.3.4 Comparison with Titian. We compare Pebble to Titian [17] since it is the only other provenance system integrated into a DISC system. The purpose of the evaluation is to compare the runtime overheads for capturing provenance of flat data items. A detailed comparison is not possible since Titian neither supports nested data, nor structural provenance, nor the programs in our scenarios. We run the test on a local machine with two worker nodes, using the unscaled articles and inproceedings records of the DBLP dataset. The test program reads each record as a long string value and filters lines containing *2015*. Then, the program computes the union over the filtered articles and inproceedings. Titian's program is implemented in the RDD API. Pebble's program is implemented in the SparkSQL API. Without provenance computation, the programs have an average runtime of 7.13 seconds and 7.36 seconds, respectively. The overall execution time is lower for the RDD program since the SparkSQL API imposes overhead on top of the underlying RDD API. Titian's overhead is 5.89%, Pebble's overhead is 6.98%.

The result indicates that for workloads on flat data supported by both systems, Pebble only adds marginal runtime overhead compared to Titian, even though it is capturing structural provenance. However, Pebble outperforms Titian in the sense that it additionally supports nested data and the collection and querying of structural provenance at attribute level.

7.3.5 Use-case analysis. To validate that structural provenance supports the use-cases described in our motivation, we revisit these use-cases with a prototypical implementation to analyze how they benefit from structural provenance.

Data-usage patterns. Pebble reveals data-access patterns, as well as hot items that frequently contribute to a query result and cold data items that do not influence any result. In Fig. 10, we show a heatmap of 25 randomly selected data items from the DBLP inproceedings dataset after running test scenarios D1 through D5. For that purpose, we merge the provenance of the individual scenarios. The more often a data item is used the redder (hot) it is. Items that do not influence any result are colored

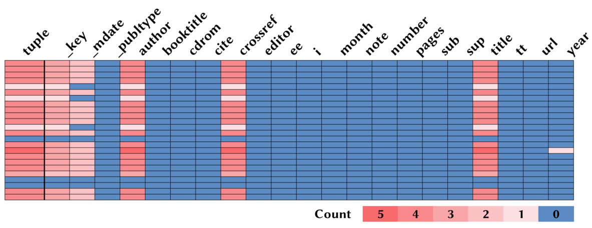


Figure 10: HeatMap for 25 data items in the DBLP inproceedings dataset after running scenarios D1-D5

blue (cold). The leftmost column indicates how often the top-level item (tuple) contributed to a result. The other columns refer to attributes of the top-level data items. The heatmap only shows top-level attributes due to space constraints. All but three top-level items have influenced at least one result. A horizontal (tuple-based) partitioning of hot and cold items, therefore, may not significantly improve system performance. However, only a fraction of all attributes contributes to the results. Thus, in this example, a vertical (column-based) partitioning of hot and cold attributes is likely to improve system performance significantly. Further, the analysis of accessed and manipulated nodes in the structural provenance reveals that the attributes *author* and *title* are frequently processed together. Thus, system performance benefits from storing these items next to each other.

In comparison, lineage solutions and PROVision [28] only provide the tuple based counter. Lipstick [2] also identifies attributes. However, Lipstick does not reveal information on access and manipulation and, thus, misses influencing attributes.

Auditing. Pebble identifies sensitive data that has been leaked directly or indirectly over the DBLP scenarios D1 through D5. All data in Fig. 10 are leaked whose count is bigger than zero. Data with count zero (blue) is not leaked. Since Pebble distinguishes access and manipulation of items, it further reveals the usage of the *year* item whose count equals one. It is marked as influencing since it does not contribute to any result item in D1 to D5. However, knowing that the *year* item is accessed is important to assess the risk of reconstruction attacks.

In comparison, lineage solutions and PROVision [28] only provide full tuples. Thus, they mark too much data as leaked. This is costly for a company, e.g., if a non-leaked (blue) attribute holds credit card numbers. Then, the company has to issue new credit cards to all marked customers, even though the information is not leaked. Lipstick [2] potentially misses leaked information, since it misses influencing attributes like the *year*. Thus, neither of the mentioned solutions allows for proper risk assessment.

8 CONCLUSION AND OUTLOOK

This paper introduced structural provenance, for which we provided a formal data model and execution semantics for operators frequently used in DISC systems. Further, we showed how to capture the structural provenance in an efficient and scalable way. Based on the captured provenance, we formalized an algorithm to backtrace structural provenance at attribute level for nested data at provenance query time. Our experimental evaluation using the Pebble system showed that our contributions result in the first DISC system integrated provenance solution for nested data that is efficient, scalable, and accurate enough to support novel provenance use-cases, such as auditing and data-usage patterns.

Future work includes extending Pebble with a user-friendly front-end to interact with structural provenance. We also intend to optimize provenance querying.

ACKNOWLEDGMENTS

Partially funded by Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy - EXC 2075 - 390740016.

REFERENCES

[1] U. Acar, P. Buneman, J. Cheney, J. Van Den Bussche, N. Kwasnikowska, and S. Vansummeren. 2010. A Graph Model of Data and Workflow Provenance. In *TaPP*.
[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB* 5, 4 (2011).
[3] Y. Amsterdamer, D. Deutch, and V. Tannen. 2011. Provenance for aggregate queries. In *PODS*.
[4] P. Buneman, S. Khanna, and W.-C. Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*.
[5] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. 1995. Principles of programming with complex objects and collection types. *Theoretical Computer Science* 149, 1 (1995).
[6] J. Cheney, A. Ahmed, and U. Acar. 2014. Database Queries That Explain Their Work. In *PDPP*.
[7] J. Cheney, L. Chiticariu, and W.-C. Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007).
[8] R. Diestelkämper. 2019. Evaluation Workload. https://www.ipvs.uni-stuttgart.de/abteilungen/de/abteilung/mitarbeiter/Ralf.Diestelkaemper_infos/resources/workload.pdf.
[9] R. Diestelkämper and M. Herschel. 2019. Capturing and Querying Structural Provenance in Spark with Pebble (Demo). In *SIGMOD*.
[10] J. N. Foster, T. J. Green, and V. Tannen. 2008. Annotated XML: Queries and Provenance. In *PODS*.
[11] M. Grabowski, J. Hidders, and J. Sroka. 2013. Representing MapReduce optimisation in the Nested Relational Calculus. In *BNCOD*.
[12] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *ICSE*.
[13] M. Hachicha and J. Darmont. 2013. A Survey of XML Tree Patterns. *TKDE* 25, 1 (2013).
[14] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB J.* 26, 6 (2017).
[15] R. Ikeda, H. Park, and J. Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *CIDR*.
[16] M. Interlandi, A. E., K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. D. Millstein, and T. Condie. 2018. Adding data provenance support to Apache Spark. *VLDB J.* 27, 5 (2018).
[17] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. 2015. Titian: data provenance support in Spark. *PVLDB* 9, 3 (2015).
[18] G. Karvounarakis and T. J. Green. 2012. Semiring-Annotated Data: Queries and Provenance. *SIGMOD Rec.* 41, 3 (2012).
[19] R. Kaushik, Y. Fu, and R. Ramamurthy. 2013. On Scaling Up Sensitive Data Auditing. *PVLDB* 6, 5 (2013).
[20] N. Kwasnikowska and J. Van den Bussche. 2008. Mapping the NRC Dataflow Model to the Open Provenance Model. In *IPAW*.
[21] M. Ley. 2009. DBLP: Some Lessons Learned. *PVLDB* 2, 2 (2009).
[22] D. Logothetis, S. De, and K. Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *SoCC*.
[23] J. Lu, T. W. Ling, Z. Bao, and C. Wang. 2011. Extended XML Tree Pattern Matching: Theories and Algorithms. *TKDE* 23, 3 (2011).
[24] J. Robie, J. Spiegel, and M. Dyck. 2017. *XML Path Language (XPath) 3.1*. W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.
[25] R. Stoica, J. J. Levandoski, and P.-A. Larson. 2013. Identifying Hot and Cold Data in Main-memory Databases. In *ICDE*.
[26] The European Parliament and the Council of the European Union. 2016. Regulation (EU) no 2016/679 (General Data Protection Regulation).
[27] Z. Wang and S. Chen. 2017. Exploiting Common Patterns for Tree-Structured Data. In *SIGMOD*.
[28] N. Zheng, A. Alawini, and Z. G. Ives. 2019. Fine-Grained Provenance for Matching and ETL. In *ICDE*.