

Universität Stuttgart

Explaining existing and missing results over nested data in big data analytics systems

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Ralf Diestelkämper
aus Mettingen

Hauptberichter: Prof. Dr. Melanie Herschel

Mitberichter: Prof. Dr. Boris Glavic

Tag der mündlichen Prüfung: 08.11.2021

Institut für Parallele und Verteilte Systeme

2021

CONTENTS

1 Introduction	15
1.1 Contributions	17
1.2 Running example	22
1.3 Thesis structure	28
2 Related Work	31
2.1 Tree-pattern matching	31
2.1.1 Tree-pattern syntax	32
2.1.2 Tree-pattern matching	33
2.2 Explanations for existing data	35
2.2.1 Provenance in big data analytics systems	37
2.2.2 Data provenance models for nested data	37
2.2.3 Feature comparison of provenance approaches	40
2.2.4 Comparison of explanations on the running example . . .	42
2.3 Explanations for missing data	44
2.3.1 Provenance-based explanations for missing data	44
2.3.2 Non-provenance-based explanations for missing data . .	46
2.3.3 Comparison of explanations on the running example . . .	47
2.4 Summary	50

3	Data model, execution model, and architecture	51
3.1	Data model	51
3.2	Execution model	58
3.3	Architecture	69
3.4	Summary	71
4	Tree-pattern matching	73
4.1	Tree-pattern syntax	74
4.2	Tree-pattern matching definition	78
4.3	Tree-pattern matching algorithm	81
4.3.1	Schema matching	82
4.3.2	Data matching	85
4.3.3	Discussion	89
4.4	Summary	92
5	Explanations for existing data	93
5.1	Structural provenance capture semantics	94
5.2	Lightweight provenance capture	109
5.3	Computing explanations for existing data	113
5.3.1	Backtracing	113
5.3.2	Discussion	127
5.4	Summary	129
6	Explanations for missing data	131
6.1	Why-not question	132
6.2	Query reparameterizations and explanations	135
6.3	Computing Explanations for missing data	143
6.3.1	Phase 1: Schema backtracing	145
6.3.2	Phase 2: Schema alternatives	147
6.3.3	Phase 3: Data tracing	150
6.3.4	Phase 4: Computing approximate explanations	167
6.3.5	Discussion	173
6.4	Summary	176

7	Implementation	179
7.1	Tree-pattern matching implementation	181
7.1.1	Schema matching	181
7.1.2	Data matching	182
7.2	Pebble and Breadcrumb implementation	184
7.2.1	Annotations	184
7.2.2	Transformations	185
7.2.3	Lazy and eager explanation computation	188
7.2.4	Rewrite optimizations and limitations	190
7.3	Summary	193
8	Evaluation	195
8.1	Evaluation setup	196
8.1.1	Cluster setup	196
8.1.2	Datasets	197
8.2	Tree-pattern matching	199
8.2.1	Workload	199
8.2.2	Schema-matching runtime	200
8.2.3	Scalability with increasing dataset size	202
8.2.4	Scalability with increasing compute resources	203
8.2.5	Runtime comparison with plain Spark	204
8.2.6	Query pipeline complexity	204
8.2.7	Discussion	206
8.3	Explanations for existing data	206
8.3.1	Workload	207
8.3.2	Runtime overhead and scalability	208
8.3.3	Space overhead	211
8.3.4	Explanation query time	213
8.3.5	Comparison with Titian	215
8.3.6	Use-cases for structural provenance	217
8.3.7	Discussion	219
8.4	Explanations for missing data	220
8.4.1	Workload	221

8.4.2 Scalability with increasing dataset size	223
8.4.3 Scalability with multiple schema alternatives	226
8.4.4 Explanation quality	229
8.4.5 Discussion	233
8.5 Summary	234
9 Conclusion & Outlook	235
9.1 Conclusion	236
9.2 Outlook	238
List of Publications	241
Bibliography	243
List of Figures	253
List of Tables	255
List of Definitions	257

ZUSAMMENFASSUNG

Die Fehlersuche in analytischen Anfragen an verteilte Systeme wie Apache Spark, Flink oder Hive ist ein aufwendiger Prozess, insbesondere wenn große Datensätze mit geschachtelten Daten analysiert werden. Um diesen Fehlersucheprozess zu erleichtern, stellen wir in dieser Arbeit neue Ansätze vor, die Erklärungen für vorhandene und fehlende Daten im Anfrageergebnis liefern. Diese Ansätze basieren auf einem formalen Daten- und Ausführungsmodell, das die Ausführungssemantik von verteilten Systemen möglichst direkt abbildet. Die direkte Abbildung ermöglicht die Erstellung von praxisnahen und relevanten Erklärungen, die beschreiben, warum Daten im Ergebnis vorhanden sind oder fehlen.

Unser erster Beitrag ist ein neuartiger, verteilter und skalierbarer Algorithmus zum Abgleich von sogenannten Tree-Patterns auf verschachtelten Daten in datenparallelen Systemen. Die Patterns ermöglichen es, geschachtelte Datenwerte beliebig zu kombinieren und somit präzise zu adressieren und abzufragen. Sie sind eine wichtige Voraussetzung, um die genannten Erklärungen für einzelne geschachtelte Datenelemente zu erhalten. Der Algorithmus gleicht ein Tree-Pattern in zwei Schritten mit den Daten ab. Er berechnet im ersten Schritt Übereinstimmungen auf dem Schema und wendet diese Übereinstimmungen in einem zweiten Schritt auf die Datenwerte an. Dadurch wird ein komplexer globaler Zustand vermieden, der

bisherige Algorithmen daran hindert, auf großen verteilten Systemen und großen Datensätzen zu skalieren.

Außerdem stellen wir neuartige Methoden zur Berechnung von Erklärungen auf geschachtelten Daten vor. Sie basieren auf Provenance. Provenance beschreibt im Allgemeinen die Herkunft und Ableitung der Ergebnisdaten einer analytischen Anfrage. Um Erklärungen für vorhandene Daten bereitzustellen, führen wir die neuartige strukturelle Provenance ein. Zusätzlich zu Datenabhängigkeiten erfasst sie strukturelle Veränderungen auf den geschachtelten Daten. Sie liefert umfassendere Erklärungen als existierende Ansätze, da sie zwischen Lese- und Schreibzugriffe auf den Daten unterscheidet und zwar so feingranular, dass sie zwischen einzelnen, geschachtelten Datenattributen unterscheidet. Wir definieren Regeln, die unser Ausführungsmodell erweitern, um die strukturelle Provenance zu erfassen. Die nach diesen Regeln erfasste strukturelle Provenance verursacht einen hohen Laufzeitzuwachs beim Ausführen der analytischen Anfrage. Deshalb stellen wir den Pebble-Algorithmus vor, der eine optimierte, leichtgewichtige strukturelle Provenance erfasst. Sie ermöglicht eine Skalierung auf große, geschachtelte Datensätze. Da Pebbles Erklärungen umfangreicher sind als die bisheriger Provenance-Lösungen, kann man Pebble für neue Anwendungsfälle jenseits der Fehlersuche einsetzen, wie beispielsweise einem Auffinden von Datennutzungsmustern oder eines feingranularen Auditing.

Darüber hinaus stellen wir in dieser Arbeit einen neuartigen Ansatz vor, der Erklärungen für fehlende Daten im Ergebnis einer analytischen Anfrage liefert. Diese Erklärungen finden Operatoren in der analytischen Anfrage, die verhindern, dass fehlende Daten im Ergebnis auftauchen, dort aber erwartet werden. Unser Ansatz ist der erste, der geschachtelte Daten unterstützt und Operatoren findet, die das Schema und die Struktur der Daten verändern, wie beispielsweise die Projektion. Außerdem berücksichtigt er fälschlicherweise referenzierte Attribute in der Anfrage. Daher sind die Erklärungen von unserem Algorithmus im Vergleich zu den Erklärungen existierender, provenance-basierter Lösungen auf eine größere Vielfalt von Datensätzen und auf neuartige Fehlerszenarien anwendbar. Um diese umfangreicheren Erklärungen zu erhalten, erweitert unser Ansatz provenance-basierte Lösungen

um Reparametrisierungen. Reparametrisierungen beschreiben Parameteränderungen in den Operatoren einer Anfrage. Wir führen diese formal ein, basierend auf unserem Ausführungsmodell, und leiten eine formale Definition unserer abfragebasierten Erklärungen ab. Um die Erklärungen effizient auf großen, verschachtelten Datensätzen zu berechnen, schlagen wir einen neuartigen heuristischen Algorithmus vor, den wir Breadcrumb nennen. Er setzt zwei neuartige Konzepte um. Erstens nutzt er sogenannte Schemaalternativen, um falsch referenzierte Attribute zu berücksichtigen, und zweitens validiert er das Zwischenergebnis nach jedem Operator in einer analytischen Anfrage erneut. Damit prüft er, welche Daten zur fehlenden Antwort beitragen können. Das ist notwendig, um korrekte Erklärungen für verschachtelte Daten zu liefern.

Wir implementieren den Algorithmus für die Tree-Patterns, den Pebble-Algorithmus, und den Breadcrumb-Algorithmus exemplarisch in Apache Spark, um zu zeigen, dass die drei Algorithmen mit wachsenden Datenmengen skalieren. Deshalb führen wir sie auf mindestens zwei verschachtelten Echtweltdatensätzen mit bis zu 500 GB Größe aus. Wir veranschaulichen außerdem, dass die Tree-Pattern die Anfragekomplexität verringern und zeigen, dass Pebble und Breadcrumb umfassendere Erklärungen liefern als andere zeitgemäße Lösungen. Dadurch können sie für neuartige Anwendungsfälle genutzt werden.

ABSTRACT

Debugging analytical queries in big data analytics systems, such as Apache Spark, Flink, or Hive is a tedious process, especially when large datasets with nested data are involved. To ease this debugging process, we present novel approaches to obtain *explanations* for existing and missing data in the query result based on a formal data and execution model that faithfully captures the execution semantics of big data analytics systems to provide practically meaningful explanations. These explanations describe why data are present or absent from the result.

Our first contribution is a novel, distributed, and scalable algorithm that matches tree-patterns on nested data in big data analytics systems. It enables us to precisely address and query nested data values and arbitrary combinations of them. We leverage this tree-pattern matching algorithm to request explanations for queries over large, nested data. The algorithm matches the pattern onto the data in two steps. It computes matches on the schema in the first step and applies these matches on the data values in a second step. Hence, it avoids complex global state that prevents other state-of-the-art algorithms to scale horizontally on large compute clusters and dataset sizes.

In addition to the tree-pattern matching algorithm, we leverage provenance to find the explanations. Provenance describes the origins and derivation of the result data. To provide *explanations for existing data*, we introduce the novel *structural provenance*. It traces structural manipulations in addition to data dependencies through the query pipelines. It provides more comprehensive explanations than other existing approaches since it distinguishes between accessed and manipulated data at the granularity of individual

nested attributes. We define formal capture rules for the structural provenance that extend our execution model. Capturing the structural provenance according to these rules imposes a high runtime overhead. Thus, we contribute the Pebble algorithm that implements an optimized, lightweight structural provenance to scale to large, nested datasets. Pebble’s explanations enable novel use-cases beyond debugging, such as finding data-usage patterns or fine-grained auditing.

Furthermore, we contribute a novel approach to *query-based explanations for missing data* in a query result. Query-based explanations pinpoint operators in the query that prevent expected data from appearing in the result. This data is called missing data or missing answer. Our approach is the first to support nested data and to consider operators that modify the schema and structure of the data such as the nesting or projection operator as potential causes of missing answers. Additionally, it accounts for mistakenly referenced attributes in the query. Hence, our explanations apply to a wider range of datasets and to novel error scenarios compared to existing, provenance-based solutions. Our approach extends these solutions with reparameterizations. Reparameterizations describe parameter modifications in query operators. We formally introduce them based on our execution model and derive a formal definition of our query-based explanations. To efficiently compute the explanations over large, nested datasets, we propose a novel heuristic algorithm, called Breadcrumb. It applies two unique techniques: (i) It reasons about multiple schema alternatives to account for the mistakenly referenced attributes and (ii) it re-validates each intermediate result to check whether its data can contribute to the missing answer. That is necessary to provide correct explanations for nested data.

We implement the tree-pattern matching algorithm, Pebble, and Breadcrumb in Apache Spark to show that each algorithm scales with increasing dataset sizes. Therefore, we run the algorithms on at least two nested real-world workloads of up to 500GB. We illustrate that tree-patterns simplify the query pipelines and show that Pebble and Breadcrumb provide more comprehensive explanations than other state-of-the-art solutions which enable novel use-cases.

ACKNOWLEDGEMENTS

First and foremost, I like to thank my advisor Melanie Herschel. She introduced me to the interesting topic of data provenance and helped me to shape and sharpen the contributions in this work. She has always had an open door when I needed advice on my research and writing. I also like to thank Boris Glavic who was a great mentor for my research, particularly for the prototype implementation. Furthermore, I thank Seokki Lee. We spent countless hours on the research on the explanations for missing data and motivated each other to get the research published. Moreover, I thank all my fellow PhD students at the University of Stuttgart who I had the pleasure to research, teach, and work with.

Finally, I like to thank my family and friends who supported me on writing this thesis in the last years. In particular, I thank my wife Indra for her patience and trust in me.

INTRODUCTION

Big data analytics systems, such as Apache Spark, Apache Flink, or Apache Hive, process vast amounts of heterogeneous data. They run on computing clusters with distributed computing nodes and provide an interface that takes SQL-like queries as input.

Such systems have become popular in recent years for multiple reasons. Among these are their ability to scale with an increasing number of compute nodes in the system and to support the processing of nested data formats. When more nodes are added to such a system, it can process bigger datasets or process the same datasets faster. Furthermore, the big data analytics systems support data formats, such as JSON, Parquet, or ProtocolBuffer, commonly used to exchange data across the internet. Unlike relational data formats, these data formats allow for nesting tuples and relations into attributes. Furthermore, these systems provide rudimentary means to transform nested tuples and relations, such as a flatten operator that unnests tuples in nested collections or a nesting operator that collects tuples into a new nested relation.

However, the systems lack means to query arbitrary nested data values and combinations of them. A well-established means to address individual

nested data values is tree-pattern matching. A tree-pattern allows to specify the structure and values of nested data [HD13]. Its nodes describe attributes in the data and may hold further constraints on attribute values. Its edges describe structural dependencies between the attributes. A tree-pattern matching algorithm applies the tree-pattern onto the nested data to obtain all data that comply with all constraints in a tree-pattern.

Furthermore, the demand to debug complex queries grows because of the increasing popularity of big data analytics systems. Therefore, multiple debugging approaches are being researched. For instance, BigDebug [GIY+16] extends Apache Spark with debugging primitives to support simulated breakpoints, watchpoints, and fine-grained latency monitoring. Other approaches, such as Titian [IST+15], Lipstick [ADD+11], Newt [LDY13], and RAMP [IPW11], capture provenance to reason about data in the query result.

In general, provenance describes any information about an end product's production process, which can be anything from a piece of data to a physical object [HDB17; MBC13]. In the context of this work, provenance describes data dependencies between input data and output data of entire queries or the individual operators within queries. These dependencies allow for reasoning about existing and missing data in the output. Existing data are part of the result, whereas missing data are not part of the result, even though they are expected.

Leveraging provenance to find explanations for existing and missing data has been the subject of prior research. The works that focus on provenance-based explanations for existing data are either of three types. (i) They are designed for flat relational data [AFG+18; MDG18] and do not trivially extend to nested data. (ii) They are designed to scale in big data analytics systems [IPW11; IST+15; LDY13] but lack provenance support for nested data. (iii) They are designed for nested data but cannot scale on big data analytics systems [ADD+11; CAA14; FGT08; ZAI19].

Research that focuses on provenance-based explanations for missing data falls into one of three categories. (i) The first group of publications computes instance-based explanations, which provide missing data in the input as

explanations [HH10; HHT09; LKLG17; LLG20; MGMS10]. (ii) The second group of publications focuses on query-based explanations [BHT14; BHT15; CJ09]. They reveal operators in a query that prevent the missing data from appearing in the queries' results. (iii) The last group of publications provides refinement-based explanations [ILZ14; TC10]. They provide information on how to modify the query to obtain the missing data. To the best of our knowledge, these existing approaches are all limited to small, relational datasets.

This thesis addresses the previously mentioned shortcomings of existing approaches to provide comprehensive query debugging means for analytical queries that process large amounts of nested data in big data analytics systems. Its overarching research question is how to leverage *provenance* to explain existing and missing query results in *big data analytics systems* when processing *nested data*. This thesis studies the problem from the ground up. Hence, it addresses the following research questions in the context of nested data: (i) How can a big data analytics system support tree-pattern matching to pinpoint and retrieve nested data in a scalable and distributed way, e.g., to identify the data to be explained. (ii) How can a system compute explanations for existing data in a more comprehensive and scalable way than existing solutions? (iii) How can a system comprehensively compute explanations for missing data in a scalable way? The next subsection summarizes our main contributions and research results that address the aforementioned questions.

1.1 Contributions

To explain existing and missing query results in big data analytics systems that process nested data, we make the following contributions:

- (1) **Data and execution model.** We define a nested data model and a nested relational algebra for bags to formally describe the explanations for existing and missing data. The nested data model supports nested

tuples and bags to faithfully depict the data structures of commonly used nested file formats such as JSON, Parquet, or ProtocolBuffer.

The nested relational algebra for bags (NRAB) describes the operators available to query the data. The operators in the algebra are closely corresponding to commonly available operators in big data analytics systems to provide meaningful explanations. For each operator, the algebra provides an inference rule that describes the operator's semantics. Together, the data model and the relational algebra allow for richer explanations than the ones provided by most other state-of-the-art solutions. We leverage them to formalize the research questions and the explanations and express the limitations of our approaches. The definitions relating to the nested data model and algebra have been published as part of [DH20b; DLHG21a; DLHG21b].

- (2) **Distributed, scalable tree-pattern matching.** We contribute a distributed, scalable tree-pattern matching algorithm to query arbitrary, nested data in big data analytics systems. Tree-patterns generally allow for precisely exploring and extracting arbitrarily nested data. We further leverage them to query and compute the explanations for nested data values.

Our matching algorithm is designed to scale horizontally on distributed computing clusters. Unlike state-of-the-art algorithms, our algorithm avoids using global state machines or computing large amounts of intermediate results. They hinder the other algorithms from scaling on distributed systems. Instead, it exploits the fact that big data analytics systems process collections of tuples that share a common schema. It splits the matching into two distinct phases: a schema matching phase and a data matching phase. In the schema matching phase, the algorithm matches the tree-pattern's nodes onto the schema to obtain schema matches. It uses the schema matches to prune the search space when matching the data values. In the data matching phase, it is indeed sufficient to check for each tuple in the input relation if it conforms to a schema match. When this is the case, the algorithm

checks further constraints defined in the tree-pattern, such as value constraints. If all constraints are satisfied, the tuple matches the tree-pattern. Since the algorithm applies each schema match on every tuple in the input relation independently, this phase scales out to the compute resources in the big data analytics system. The tree-pattern matching algorithm has been published in [DH20a].

- (3) **Explanations for existing data.** We use the tree-patterns to request explanations for data existing in a result and propose the novel concept of structural provenance that captures the access to and the manipulation of nested data in addition to mere data dependencies. That allows for fine-grained explanations and efficient provenance capture at the same time. Existing provenance solutions for nested data only trace manipulated data and lack scalability to large datasets.

We formally extend our algebra's operators with capturing rules for structural provenance annotations. The annotations are sufficient to compute explanations for arbitrary nested data elements but hold redundant information. Therefore, we introduce a lightweight version of the structural provenance annotations that reduces the annotation overhead to the amount of state-of-the-art solutions that exclusively capture annotations for flat data.

We implement the lightweight structural provenance in the Pebble algorithm. It captures the lightweight annotations during query pipeline execution and computes explanations for nested data when an explanation for selected data in the result is requested. These explanations carry information on the accessed and manipulated data at the granularity of individual nested data values. We illustrate that the explanations enable novel use-cases beyond debugging such as identifying data-usage patterns or comprehensive auditing analytics. We further show that Pebble scales to large dataset sizes. Work relating to Pebble has been published in [DH19; DH20b]

- (4) **Explanations for missing data.** In addition to the explanations for existing data, we contribute a novel approach to query-based explana-

tions for missing, nested data. Existing, provenance-based approaches are limited to flat relational data. Hence, our approach supports a wider variety of input data formats. Furthermore, existing approaches exclusively find selective operators that remove the missing data from the output. Our approach extends them with the novel concept of reparameterizations. Reparameterizations reflect any changes to operator parameters in a query. Therefore, our explanations reveal more error scenarios than previous approaches, such as misinterpreted attributes that prevent data to appear in the result.

We formally introduce the nested reparameterizations and explanations based on our nested algebra to define allowable reparameterizations. To avoid superfluous operator reparameterizations, our explanations only consider query reparameterizations that modify exactly the operators needed to obtain the missing data in the result. To avoid unnecessary changes to the query result, the explanations further consider side-effects on the query result. Side-effects are all data that appear or disappear in the result in addition to the expected data after reparameterizations. Given a tree-pattern that describes the missing data, we define successful reparameterizations as reparameterizations that yield the missing answer. They are minimal if no other successful reparameterization exists that modifies a subset of its operators and, additionally, has fewer side-effects on the query result. Explanations correspond to the set of reparameterized operators in the minimal successful reparameterizations.

Finding these explanations is computationally infeasible. Therefore, we propose the scalable, heuristic Breadcrumb algorithm that leverages schema alternatives and revalidation to approximate the explanations for a practically relevant set of queries. The schema alternatives represent sets of reparameterizations that replace attributes in the query with alternative attributes. They allow Breadcrumb to trace large amounts of reparameterizations simultaneously. The revalidation ensures that Breadcrumb does not mark data as potentially contributing

to the missing result that cannot produce the missing result. Based on these concepts, Breadcrumb provides larger numbers of more comprehensive query-based explanations than state-of-the-art solutions. It computes them on nested datasets that are 1000+ times larger than the datasets used to evaluate previous solutions. We have published our research on Breadcrumb in [DGHL19; DLGH21; DLHG21a; DLHG21b].

- (5) **Implementation and evaluation.** To evaluate the tree-pattern matching algorithm, the Pebble algorithm, and the Breadcrumb algorithm, we integrate them into Apache Spark. While Apache Spark is our big data analytics system of choice the concepts introduced in this work also apply to other big analytics systems, such as Apache Flink, Apache Hive, and Apache Pig.

We conduct multiple experiments on at least two real-world datasets of up to 500GB size to show that the three implemented algorithms scale with increasing dataset sizes on distributed compute resources. We further conduct algorithm-individual experiments to measure algorithm-specific runtime or space properties.

Furthermore, we show that the tree-pattern matching algorithm significantly simplifies the query pipeline definition. Pebble provides such comprehensive explanations that it enables novel use-cases such as auditing or identifying data-usage patterns. Breadcrumb yields more comprehensive explanations than other state-of-the-art solutions. We can even show that it is the only query-based solution that finds correct and complete explanations in multiple real-world scenarios. The results have been published alongside the algorithms in the publications cited above.

The five contributions directly address this thesis' research question on explaining existing and missing query results on nested data in big data analytics systems. Contribution (1) provides the formal data and execution model used to formally describe and scope the problem of our explanations.

Contribution (2) directly addresses the research question (i). Contributions (3) and (4) leverage the tree-patterns from Contribution (2) to query the explanations. Contribution (3) provides explanations for existing data and addresses research question (ii). Contribution (4) yields explanations for missing data. Hence, it answers research question (iii). Contribution (5) emphasizes on the scalability of Contributions (2) to (4) and the explanation quality of Contributions (3) and (4). Together, all five contributions provide a detailed answers to the thesis’ overall research question.

Next, we introduce a running example that illustrates the explanations for existing and missing data. Furthermore, it motivates the need for a scalable tree-pattern matching algorithm, since Pebble and Breadcrumb need to leverage it to compute the explanations.

1.2 Running example

To illustrate our contributions throughout this work, we introduce a running example based on nested customer data. Table 1.1 shows five customers with their `firstname` and `lastname` and the two address attributes `address1` and `address2`, which could represent the billing and shipping address along with a timestamp of their last usage. The address attributes hold nested relations that hold collections of tuples with a `city` a `year` attribute. For conciseness, we limit the example addresses to the mentioned two attributes. The *italic numbers* serve as tuple identifiers. They do not belong to the actual input data.

The company’s retail analysts like to compute all cities to which the company delivered goods in 2019. For each of these cities, they obtain a

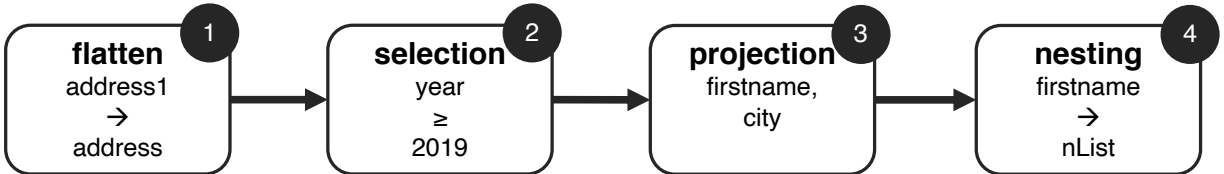


Figure 1.1: Example query pipeline

	firstname	lastname	address1		address2	
1	Peter	Jones	city	year	city	year
			LA	2010	LA	2019
			SF	2018	LV	2018
					SF	2017
2	Sue	Miller	city	year	city	year
			LA	2019	NY	2019
			NY	2018	LA	2018
3	Sue	Walker	city	year	city	year
			SF	2018	LV	2018
			LA	2019	NY	2019
4	Tom	Smith	city	year		
			LA	2019		

Table 1.1: Example input data

	city	nList
101	LA	firstname
		Sue
		Sue
		Tom

Table 1.2: Example output data

nested relation of customer names. For that purpose, they define the big data analytics query in Figure 1.1. The four boxes describe operators. The numbers in circles represent the unique operator ids. The query pipeline flattens the nested relation address1 out and filters by the year. Then, it selects only the firstname and city to create a nested relation nList of the firstname for each city.

The query's result is shown in Table 1.3. It contains only one tuple with the city LA. This tuple has a nested relation of the three firstnames: Sue,

Sue, and Tom. The company’s retail analysts did not expect this result for two reasons.

(i) They wonder why Sue occurs twice in the nested relation nList of city LA as highlighted in the result Table 1.3.

(ii) One of the analysts further remembers that the company has shipped to NY, too. However, NY does not appear in the result. Thus, the analysts expect a tuple like the one shown in Table 1.4. The asterisk represents a placeholder for arbitrary tuples in the nested relation.

city		nList				
101	LA	<table border="1"> <thead> <tr> <th>firstname</th> </tr> </thead> <tbody> <tr> <td>Sue</td> </tr> <tr> <td>Sue</td> </tr> <tr> <td>Tom</td> </tr> </tbody> </table>	firstname	Sue	Sue	Tom
firstname						
Sue						
Sue						
Tom						

Table 1.3: Unexpected data that is **existing** in the example output

city		nList		
∅	NY	<table border="1"> <thead> <tr> <th>firstname</th> </tr> </thead> <tbody> <tr> <td>*</td> </tr> </tbody> </table>	firstname	*
firstname				
*				

Table 1.4: Expected data that is **missing** in the example output

The first question can be answered with an explanation for existing data and the second one with an explanation for missing data. To obtain these explanations the analysts first need to concisely address the highlighted data in Table 1.3 and in Table 1.4. However, state-of-the art big data analytics systems such as Apache Spark, Flink, or Hive lack built-in means to address them accurately. Tree-patterns would enable the analysts to accurately address the highlighted attributes. Therefore, we contribute an integrated tree-pattern matching algorithm that matches tree-patterns as the ones shown in Figure 1.2 onto nested relations as the one shown Table 1.2.

As listed in Figure 1.2c, the tree-patterns have nodes that represent attributes except for the root node. This node matches any top-level tuple and serves as the entry point to its attributes. Edges either represent parent-child relationships between attributes when they are single-lined or ancestor-descendant relationships when they are double-lined. Parent-child edges require the child to be a direct descendant of the referenced parent. In contrast, ancestor-descendant edges only require a descendant with the

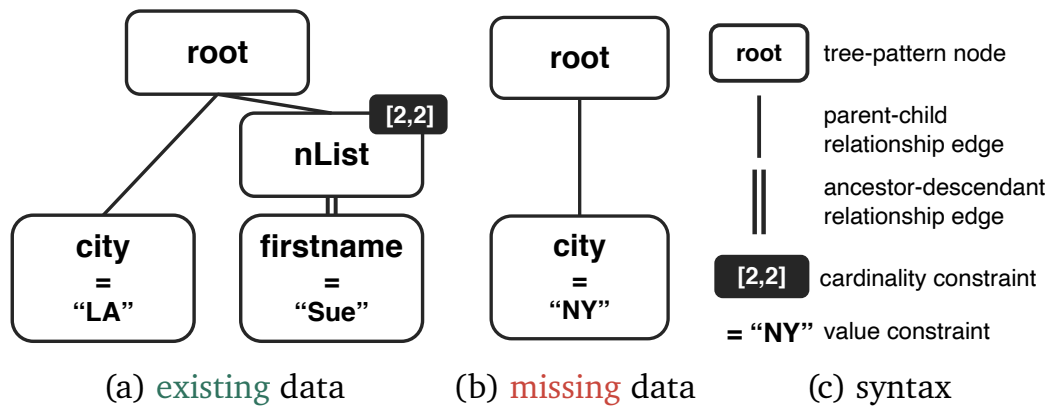


Figure 1.2: Tree-patterns that match the corresponding data in Table 1.3 and Table 1.4 with syntax legend

defined attribute name in the ancestor’s nested attributes. The number of attributes between the ancestor and descendant does not matter. Black boxes impose cardinality constraints over the associated node. They only apply to attributes of nest relation types. Nodes only match the data if the node’s subtree occurs n times in the data. The number n has to be in the interval defined in the black box. Conditions in the node impose value constraints over the data. They only match if the according attribute satisfies the condition in the node.

The tree-pattern in Figure 1.2a matches exactly the highlighted data in Table 1.3. A look at the pattern’s left branch shows that the city node in Figure 1.2a matches the city attribute in Table 1.3 for two reasons. First, the node name equals the attribute name. Second, the attribute’s value is LA, as required by the city node’s value constraint. Analogously, the firstname node in the right branch matches on both occurrences of the name Sue in the nested nList. Since there are two occurrences, they also satisfy the cardinality constraint defined on the nList node.

In contrast to the tree-pattern in Figure 1.2a, the tree-pattern in Figure 1.2b must not match onto a tuple in Table 1.2 since it describes an expected but missing result. However, it concisely describes the missing tuple in Table 1.4, which requires the city to be NY.

	firstname	lastname	address1		address2	
1	Peter	Jones	city	year	city	year
			LA	2010	LA	2019
			SF	2018	LV	2018
					SF	2017
2	Sue	Miller	city	year	city	year
			LA	2019	NY	2018
			NY	2018	LA	2019
3	Sue	Walker	city	year	city	year
			SF	2018	LV	2018
			LA	2019	NY	2019
4	Tom	Smith	city	year		
			LA	2019		

Table 1.5: Explanations for the **existing** example output, with **contributing** and **influencing** data

We contribute the Pebble and Breadcrumb algorithms. Given the tree-patterns in Figure 1.2, the algorithms precisely point the analysts to the causes for the duplicate Sue in the nested relation and the missing city New York in the result.

The novel Pebble algorithm that implements the structural provenance yields the highlighted data in the input Table 1.5. Data are highlighted in two colors. The **dark green color** describes contributing data and the **light green color** influencing data. In the example, the two firstnames Sue and the city entries LA, are highlighted as contributing since they are precisely those values that yield the highlighted values in Table 1.3. Furthermore, the Pebble algorithm highlights those year values as influencing that are associated with the city LA because the selection in Figure 1.1 accesses the highlighted years to filter on them.

When the retail analysts look at the explanation, they immediately understand that Sue occurs twice in the result because this firstname belongs to two different persons who share a common firstname and city. Thanks to

explanations		
<table border="1"> <thead> <tr> <th>operator</th> </tr> </thead> <tbody> <tr> <td>2</td> </tr> </tbody> </table>	operator	2
operator		
2		
<table border="1"> <thead> <tr> <th>operator</th> </tr> </thead> <tbody> <tr> <td>1</td> </tr> </tbody> </table>	operator	1
operator		
1		

Table 1.6: Explanations for the **missing** example output

Pebble, the analysts did not need to conduct any manual debugging to understand the root cause for duplicate entry in the result. They saved time and money. Pebble’s structural provenance further reveals that the year influences the result even though it not exposed in it. This information is not needed to understand the duplicate Sue in the result. However, for instance, it helps in understanding which personal information the query reveals about the customers. Here, both Sues have been recent customers. This small example already illustrates potential applications for structural provenance beyond debugging. In this work, we illustrate further applications for structural provenance.

Satisfied with the explanation for the duplicate Sue in the result, the analysts investigate why the city NY is absent from the result. Thus, they apply the novel Breadcrumb algorithm, which yields the list of operators in Table 1.6. The numbers refer to the operator identifiers in the query pipeline in Figure 1.1. Each entry in the table describes one explanation that makes NY appear in the result. The first result in the table indicates that NY appears in the result if the analysts modify the selection operator with identifier 2. If they replaced the constant year 2019 with 2018, for instance, NY appears in the result. The second result describes that the flatten operator with identifier 1 requires modification. If the analysts replace attribute address1 with address2 in the depict flatten operator NY also appears in the result. Hence, both explanations are correct, because operator reparameterizations exist that make the city New York appear in the result.

Eventually, the retail analysts realize that they have misinterpreted `address1` as the shipping address. However, it is the billing address. Actually, `address2` is the shipping address. They modify the flatten operator accordingly and obtain the desired query pipeline. Once again, the analysts saved time and money when debugging the query, since Breadcrumb pointed them directly to the operators of interest.

This example is kept small on purpose to illustrate the ideas behind the concepts and algorithms contributed in this thesis. The concepts and algorithms particularly play out their full potential on large nested datasets with wide schemata and multiple nesting layers as we highlight throughout the thesis. Next, we provide an overview of the thesis' structure.

1.3 Thesis structure

This work is structured in multiple chapters that gradually introduce our contributions in detail.

Chapter 2. In this chapter, we relate our contributions to state-of-the-art work. In particular, we compare our tree-pattern syntax to other syntaxes. We also describe our tree-pattern matching algorithm's unique features that make our algorithm scale on big data analytics systems. Additionally, we describe provenance solutions for nested data and big data analytics systems and highlight how they differ from the structural provenance and Pebble algorithm. Moreover, we compare the reparameterization-based explanations and the Breadcrumb algorithm to related approaches that explain missing data.

Chapter 3. To address Contribution (1), we introduce a formal nested data model and a nested relational algebra for bags to transform the nested data. Furthermore, we provide an architecture overview, that illustrates the interplay between the tree-pattern matching algorithm and the Pebble and Breadcrumb algorithms.

Chapter 4. In this chapter, we formally introduce the tree-patterns and the tree-pattern matching algorithms. We first describe the tree-pattern

syntax and matching formalism, followed by a detailed description of our tree-pattern matching algorithm. This chapter focuses on Contribution (2).

Chapter 5. Once we have defined the tree-pattern matching algorithm, we provide details on the explanations for existing data to address Contribution (3). We formally introduce the structural provenance and describe the Pebble algorithm. It computes explanations for existing data in the result. We define inference rules that describe the provenance collection. We show how to optimize the collection for scalability reasons. Further, we describe the backtracing algorithms needed to compute the explanations from the collected provenance.

Chapter 6. Afterward, we focus on the query-based explanations for missing data to address Contribution (4). We formally introduce operator reparameterizations, successful reparameterizations, and minimal successful reparameterizations to define the query-based explanations for missing data. Since computing these explanations is NP-hard in the general case, we propose the heuristic Breadcrumb algorithm, which approximates the explanations.

Chapter 7. We implement the three novel algorithms into the a big data analytics system Spark and describe our implementation decisions. We emphasize on optimizations that allow our algorithms to scale to large datasets and describe implementation limitations. This chapter is part of Contribution (5).

Chapter 8. To demonstrate that our algorithms scale to large dataset sizes in big data analytics systems, we conduct experiments on two real-world workloads of up to 500GB. We further assess the explanation quality of the Pebble and Breadcrumb algorithms. We sketch use-cases and applications for our algorithms beyond debugging. This chapter mainly addresses Contribution (5). It further reinforces Contributions (2) to (4).

Chapter 9. In the last chapter, we summarize our work and provide an outlook on future work.

Given the motivation, the research questions, the contributions, the running example and overview of the thesis structure, we introduce the related work in the next chapter.

RELATED WORK

The tree-pattern matching algorithm and the explanations for existing and missing data are related to existing works. In this chapter, we show similarities and differences between them and our work to highlight our contributions. It extends and updates our survey [HDB17] and the related work sections of our published papers on tree-pattern matching [DH20a], explanations for existing data [DGHL19; DH20b], and explanations for missing data [DLGH21; DLHG21a; DLHG21b]. First, we compare existing tree-pattern research with our tree-pattern matching algorithm. Next, we distinguish our provenance-based explanations for existing data from other provenance systems for big data analytics systems and provenance models for nested data. Finally, we distinguish our work on missing data from existing work that explains missing data.

2.1 Tree-pattern matching

Multiple surveys have summarized the state-of-the-art research for tree-patterns and tree-pattern matching [HD13; LLBW11; TPL+13; TTT19]. Hence, we keep our comparison with related tree-pattern research short.

2.1.1 Tree-pattern syntax

According to Hachicha et al. [HD13], the expressiveness of tree-patterns varies in multiple properties. These properties are either defined for the tree-pattern's edges or nodes. Few tree-patterns only support parent-child edges; others only support ancestor-descendant relationships. Most tree-patterns support both edge types like our tree-pattern syntax. Further, state-of-the-art tree-patterns for XML data also support optional edges and absent edges. Optional edges indicate that the linked descendant nodes can match the data, but they do not need to match. Absent edges enforce that a match on the data must not have the linked descendant nodes. Our tree-pattern syntax supports neither of these two edge types because the data model of big data analytics systems does not allow for optional schema elements in the data. These systems process relations of tuples that share the same schema. Thus, an edge is either present for all tuples or no tuples, making optional and absent edges superfluous.

The nodes in the tree-patterns typically describe attribute names, like in our tree-pattern syntax. The nodes can further be associated with a boolean expression to constrain the value on matching data. The node only matches the data if the boolean expression over the data value evaluates to true. Our tree-pattern syntax supports these boolean expressions since they are needed to pinpoint data values of interest.

Rigid tree-patterns enforce order upon siblings. Siblings are nodes that share the same parent node. They only match the data if the attributes' order in the data is the same as the siblings' order in the tree-pattern. Our tree-patterns ignore the ordering of siblings because the names of sibling attributes must be unique in big data analytics systems. Thus, the enforced node order does not increase the expressiveness of the tree-pattern.

Furthermore, tree-patterns may support logical nodes and wildcard nodes. Logical nodes describe logical operators, such as AND, OR, or NOT and impose logical relationships between their subtrees. For instance, the logical OR operator implies that at least one of the subtrees must match the data, which one does not matter. Our tree-pattern syntax does not support logical

nodes since they are mainly syntactical sugar that our system does not need to compute provenance-based explanations. Wildcard nodes match any data. While our system generally does not support wildcard nodes, we add wildcard nodes to the tree-patterns that query explanations for missing data. That allows us to express intricate patterns with deep nesting efficiently.

Tree-pattern nodes may also come with associated cardinality annotations. They describe how often the node or its subtree has to occur in the data to obtain a match. Our tree-pattern syntax supports cardinality annotations since this is important for debugging. For instance, it makes debugging duplicates in nested relations efficient.

In summary, our tree-pattern syntax supports precisely those syntax elements that make them powerful enough to support sophisticated debugging scenarios over nested data processed by big data analytics systems. To match our tree-patterns onto the data, we apply a novel tree-pattern matching algorithm tailored to the distributed data processing of big data analytics systems.

2.1.2 Tree-pattern matching

We distinguish our matching algorithm from existing algorithms. These focus either on minimizing the tree-pattern size or on optimizing the data access [HD13].

Since an increasing tree-pattern size leads to an increasing effort to match the tree-pattern onto the data, noticeable works on tree-pattern matching [ACLS01; MS02] aim at minimizing the tree-pattern. They show that minimizing the pattern reduces the matching time on the data. While this work is a promising path to pursue, our algorithm focuses on optimizing data access.

Those algorithms that optimize the data access have two phases. The first phase applies a labeling scheme on all nodes in the data before the second phase computes the matches based on the labels [HD13]. The algorithms divide into two approaches: Structural join approaches and holistic twig join approaches. Structural join approaches typically apply a region encoding

on the data, which indicates each data value's position in the document. After labeling, these approaches split the tree-patterns into smaller parts, match these smaller parts against the data, and merge them to a complete tree-pattern match [AJK+02; ZND+01]. They have the caveat that they produce a large amount of potentially unnecessary intermediate results.

Holistic twig join approaches aim at reducing the massive amounts of intermediate results. Therefore, they apply more effective labeling schemes, such as Dewey labeling, that encodes the root-to-leaf paths and the nesting depth to identify and index the data more efficiently than region encoding [LLCC05; LML11]. Furthermore, they make use of global data structures [GBH10], such as stacks [BKS02; LCL04] and state machines [LLCC05; LML11]. The TwigVersion [WL08] algorithm further identifies repetitive structures in the nested data to improve the tree-pattern matching. The S^3 [IHH09] algorithm extends this idea. It creates a QueryGuide structure that extracts all paths from the nested data. Then it applies a matching on these paths before accessing the actual data.

The algorithms that optimize the data access are all designed for matching tree-patterns on a single XML document on a stand-alone computer. They either produce a large amount of potentially unnecessary intermediate results or make use of global data structures. Both properties are unacceptable for distributed execution. Distributing large amounts of intermediate results across distributed computing resources leads to network contention and, thus, to low overall performance. Maintaining global data structures inhibits the algorithms to scale horizontally across the computing resources because of lock contention on the global data structures. Notably, TwigVersion and S^3 consider constraints on the schema and the data separately. However, they still operate on the complete input XML document, labeling and indexing each element in the data.

In contrast to the existing algorithms, our approach scales on distributed clusters since our algorithm only applies labeling on the common schema that all top-level tuples in the input data share. Furthermore, our algorithm can directly access the nested data, given the labeled schema. It does not require

indexing each element in the data. Hence, it neither requires synchronized global data structures nor yields extensive intermediate results.

Our solutions that compute the explanations for existing and missing data in the result leverage the tree-pattern matching algorithm to compute the explanations. We discuss related work on explanations for existing data next.

2.2 Explanations for existing data

This section provides an overview of related research on provenance and distinguishes our work from existing research results. Our survey [HDB17] describes a provenance hierarchy with four potentially overlapping layers that categorize provenance research by their provenance models and provenance instrumentations. The two layers with the most specific provenance model and the highest degree of instrumentation are particularly relevant for this work: workflow provenance and data provenance. Workflow provenance only applies to workflows with a directed graph structure. In this graph, nodes represent arbitrary functions or modules with input, output, and parameters; and edges describe dataflow or control flow. Data provenance is even more restrictive. It tracks the processing of individual data items, such as tuples or data values. It typically applies to structured data models and declarative query languages with clearly defined operator semantics.

We point out the similarities and differences between our work and existing work on workflow and data provenance closely related to our work. First, we provide an overview of workflow provenance to put our work into the context of workflow provenance. In particular, we introduce and compare related workflow provenance solutions for big data analytics systems. Next, we discuss related data provenance solutions that explain existing results. We put focus on solutions that handle nested data. Finally, we discuss data provenance solutions that explain missing answers.

Solution	Granularity		Form		
	Coarse	Fine	Prospective	Retrospective	Evolution
Data Analytics					
Differential DF [CLMR16]	✓	✓		✓	
HadoopProv [ASH13]	✓	✓		✓	
Inspector Gadget [OR11]	✓	✓		✓	
Lipstick [ADD+11]	✓	✓		✓	
Newt [LDY13]	✓	✓		✓	
RAMP [IPW11]	✓	✓		✓	
SAMbA [GMF+20; GSM+18]	✓	✓	✓	✓	
Titian [IES+18; IST+15]	✓	✓		✓	
Pebble [DH19; DH20b]	✓	✓	✓	✓	

Table 2.1: Overview of big data analytics provenance solutions (adapted from [HDB17]; updated)

In our survey [HDB17], we have created a categorization of workflow provenance. It has three dimensions based on the application *domain*, the provenance *granularity*, and the provenance *form*.

We identify four major domains: business, science, data analytics, and general programming. Concerning the granularity of captured provenance, we consider provenance solutions from coarse-grained to fine-grained provenance. Coarse-grained provenance captures dependencies between input datasets and output datasets in their entirety. Fine-grained provenance captures dependencies between individual elements in the input and output datasets. Regarding the form of provenance, we distinguish retrospective, prospective provenance, and evolution provenance. Retrospective provenance captures information on the workflow’s execution. Prospective provenance captures information on the workflow’s structure that is independent of the workflow’s execution. Evolution provenance captures modifications of the workflow definition itself. In this work, we focus on fine-grained provenance in data analytics and refer the interested reader to the surveys [FKSS08; Gla21; HDB17; Mor10; PFMB19; RESC15] for research on workflow provenance in other domains.

2.2.1 Provenance in big data analytics systems

In big data analytics systems, multiple provenance solutions that compute explanations for existing data have emerged in the past ten years. Table 2.1 provides an overview of such provenance solutions and the granularity and form they provide. Our solutions are highlighted in bold text at the bottom of the table. Regarding the supported granularity, all of the listed solutions capture the provenance of individual tuples in the processed data. Thus, they all collect fine-grained provenance.

Further, all the solutions in Table 2.1 capture retrospective provenance about the processed data. Notably, only SAMbA [GMF+20; GSM+18], our Pebble algorithm explicitly exploit prospective provenance. SAMbA employs the prospective provenance for two purposes. First, it uses prospective provenance to support so-called black-box operators. Black-box operators are operators whose internal behavior is not precisely defined or even known. In big data analytics systems, typical examples for black-box operators are user-defined functions or integrated library code. Second, SAMbA employs prospective provenance to illustrate the big data pipeline, including operator parameters in their web-based debugging interface. In contrast, Pebble exploits prospective provenance to minimize the amount of collected retrospective provenance needed to trace nested data accurately. For instance, it captures operator parameters, such as a filter condition to record transformations on the schema rather than for each tuple in the dataset.

2.2.2 Data provenance models for nested data

The transition from fine-grained workflow provenance to data provenance is seamless. Some of the provenance solutions mentioned in the previous section capture provenance that comply with formal provenance models - at least to some extent. Here, we discuss different kinds of provenance models for nested data and their applicability to big data analytics systems. At least three significant 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.

The provenance models for why-, how-, and where-provenance are typically based on provenance polynomials. The polynomials formally describe the dependencies between the output and the input data. The why-provenance polynomials captures the input data that witness the existence of result tuple [CCT09]. The how-provenance polynomials further carry information on how a query derives a result tuple from the input data [CCT09]. Where-provenance captures where a data value in the result is copied from [CCT09]. We summarize the research on models for why-, how-, and where-provenance as follows. For unions of conjunctive queries, Buneman et al. [BKT01] define a why- and where-provenance model for nested data. This model does not extend to the programs defining data analytics pipelines in big data analytics systems, like the one shown in Figure 1.1, since they may include map or reduce functions or any other higher-order functions in general. To model the how-provenance of nested data, Foster et al. [FGT08] and Karvounarakis et al. [KG12] propose a semiring-model for a small subset of XQuery operators. 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 [ADT11]. PROVision [ZAI19] applies a how-provenance model for relational data and accounts for nesting and unnesting by nesting and unnesting the associated provenance expression, respectively. This model is not capable of tracking individual nested data values as the other mentioned models. All mentioned models have in common that they only capture dependencies of data values. They do not explicitly capture structural manipulations on the data and do not distinguish between access and manipulation of data.

Multiple contributions rely on graph-based provenance models. Lipstick [ADD+11] makes use of a graph model to describe the how-provenance where possible. It lacks a formal model definition for aggregations, nesting, and flattening of nested data. For these operators, it only tracks dependencies between data values. Kwasnikowska and Acar et al. [ABC+10; KV08] also

employ provenance graphs to model data dependencies. Their models are essentially limited to the operations defined in the Nested Relational Calculus (NRC) [BNTW95], which do not include aggregations or joins. In summary, these graph-based provenance models only capture provenance at a high degree of detail on simple queries, e.g., without aggregations. Then, the models even allows for capturing data structure and dependencies between data values.

In most aspects, program slicing is closest to our structural provenance model. Program slicing is a frequently used method in general programming to analyze dependencies between individual instructions [Gla21]. Applied to big data analytics pipelines, this method captures fine-grained provenance for nested data. In that respect, the work closest to our structural provenance model is Cheney et al.'s program slicing model [CAA14], which tracks provenance traces for nested relational calculus operators over nested data. The model is limited to a small set of semantically fully specified NRC operators to provide formal guarantees. 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 rather than structural manipulations. It is not expressive enough to faithfully capture and query structural manipulations.

In contrast to the existing models, our structural provenance model is tailored to big data analytics systems since it captures changes in the data structure and the data values. It distinguishes between access and manipulation of data items to capture contributing and influencing data items simultaneously, of which none of the existing approaches is capable. Furthermore, we follow a best-can-do approach for higher-order functions supporting user-defined functions, such as a map-operator, rather than not supporting them at all. While possibly not achieving the most precise provenance for some operators, this makes our approach support a broader set of big data analytics queries compared to previous systems. Finally, our model is the first to allow structural query processing explicitly.

To concisely illustrate the similarities and differences between our structural provenance and other formal provenance models for nested data and between Pebble and other provenance solutions for big data analytics systems we conduct a feature comparison.

2.2.3 Feature comparison of provenance approaches

We compare structural provenance as implemented in Pebble to the provenance solutions from Section 2.2.1 and the provenance models from Section 2.2.2 to show the range of supported features. Table 2.2 and Table 2.3 provide an overview of the provenance solutions from Section 2.2.1 and the models from Section 2.2.2, respectively. In both tables, Pebble is in the rightmost column for convenience. If another solution, such as Lipstick [ADD+11], is a big data analytics solution and comes with its own provenance model, it only appears in Table 2.2 to preserve clarity.

Since we have discussed the first three features listed in the tables in the previous sections, we focus on the last five features in this section. Pebble can capture provenance eagerly and lazily. Eager provenance is active when the big data analytics pipeline is executed. In contrast, lazy provenance is only active at provenance query time. It imposes no overhead on the actual query execution. Depending on the provenance application, either of the approaches has advantages. Like Pebble, Cheney et al.’s program slicing [CAA14] supports both ways to query the provenance. Most provenance solutions for big data analytics systems only support eager provenance computation since they prevent the entire query from re-executing. Especially in the context of big data, that may be prohibitively expensive. A notable exception is PROVision [ZAI19]. It collects provenance lazily in an external provenance solution that runs in memory.

Feature	Differential DF [CLMR16]	HadoopProv [ASH13]	Inspector Gadget [OR11]	Lipstick [ADD+11]	Newt [LDY13]	PROVision [ZAI19]	Ramp [IPW11]	SAMbA [GMF+20]	Titian [IST+15]	Pebble [DH20b]
Data provenance for nested data	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓
Provenance of acces and manipulation	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Provenance of data item structure	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Eager/lazy provenance computation	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✓
Implementation-independent provenance query formalism	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓
DISC system compatibility/integration	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✗/✓	✓/✗	✓/✗	✓/✓	✓/✓
Reported implementation	Differential Dataflow	Hadoop	PigLatin	PigLatin	Hadoop/Hyracks	Java	Hadoop	Spark RDDs	Spark RDDs	Spark Datasets
Evaluated for scalability	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓

✗ Not Supported
✓ Supported

Table 2.2: Feature overview of provenance solutions for big data analytics systems (from [DH20b]; updated and extended)

Feature	HowProv Nested [FGT08]	Why/Where Prov [BKT01]	Kwasnikowska [KV08]	Acar [ABC+10]	Program Slicing [CAA14]	Pebble [DH20b]
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.	n.a.	n.a.	n.a.	✓
DISC system compatibility/integration	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✓/✓
Reported implementation	no	no	no	Haskell	Haskell	Spark Datasets
Evaluated for scalability	✗	✗	✗	✗	✗	✓

✗ Not Supported
✓ Supported

Table 2.3: Feature overview of provenance models for nested data (from [DH20b]; updated and extended)

Furthermore, only Lipstick [ADD+11], SAMbA [GMF+20; GSM+18], and Pebble [DH20b] provide an implementation-independent provenance query formalism. Lipstick and SAMbA both provide a visual exploration interface to navigate through the provenance graph they capture during query execution. While this interface is nice to explore data dependencies between data values, it lacks the means to systematically address and trace the structure of nested data and arbitrary combinations of data values. In contrast, Pebble provides a tree-pattern interface that overcomes the mentioned shortcomings. It allows users and applications to query arbitrary combinations of nested data values and structures effectively.

In contrast to the provenance models from Table 2.3, the solutions from Table 2.2 are all at least compatible with one big data analytics system. However, only Newt [LDY13], Ramp [IPW11], and Titian [IST+15] provenly scale to data sizes beyond a few Gigabytes. These three solutions only support flat data. Thus, they are incapable of tracking nested data elements precisely. On the contrary, those solutions and models that support nested data have never been evaluated on noticeable amounts of data because they require to annotate each nested item individually [ADD+11]. Pebble scales like Newt, Ramp, and Titian to large data sizes while providing the detailed structural provenance for nested data.

We apply Pebble and other existing solutions on our running example to illustrate why it is important to precisely track nested data on large datasets.

2.2.4 Comparison of explanations on the running example

To illustrate the benefits of a scalable provenance solution for big data analytics systems that faithfully captures structural provenance for nested data, we compare the explanations that different provenance solutions provide on the running example from Section 1.2. For that purpose, we show the input data in Table 2.4 again. This time, we do not only highlight Pebble's explanations but also those of Newt [LDY13], Ramp [IPW11], Titian [IST+15], PROVision [ZAI19], and Lipstick [ADD+11].

	firstname	lastname	address1		address2	
1	Peter	Jones	city	year	city	year
			LA	2010	NY	2010
			SF	2018	LA	2019
					LV	2017
2	Sue	Miller	city	year	city	year
			LA	2019	NY	2019
			NY	2018	LA	2018
3	Sue	Walker	city	year	city	year
			SF	2019	LV	2017
			LA	2018	NY	2019
4	Tom	Smith	city	year		
			LA	2019		

Table 2.4: Example input data with highlighted explanations. The color coding indicates **contributing explanations**, **influencing explanations**, **nesting-aware tuple-based explanations**, and **non-nesting-aware tuple-based explanations**.

As mentioned in Section 1.2, our solution identifies the **contributing data** and the **influencing data**. Lipstick identifies only the **contributing data**. Thus, it lacks crucial information for applications beyond debugging, such as identifying indirectly leaked personal data as described in Section 1.2. PROVision supports tuple-based provenance that is aware of nested data. Thus, it yields the entire **tuples 2 and 3** as an explanation. In the running example, that is no problem since the dataset is small. However, PROVision is unable to identify nested elements in the input data. If the nested lists have hundreds of nested tuples as in the real-world Twitter dataset, which we use for our evaluation, identifying the correct nested data becomes a tedious manual task. Newt [LDY13], Ramp [IPW11], Titian [IST+15] only support non-nesting-aware tuple-based provenance. Thus, they yield **tuple 4** in addition to **tuples 2 and 3** as an explanation. More generally, they yield all tuples that contributed to the result tuple. Hence, in large nested data,

the meaningful part of the explanations may become occluded by irrelevant explanations.

In the context of explanations for existing data, we have distinguished our structural provenance and the Pebble algorithm from other provenance approaches and showed their explanations on the running example. The example illustrates that Pebble provides more comprehensive explanations than other existing solutions. Next, we discuss the solutions that compute explanations for missing data.

2.3 Explanations for missing data

Little more than a decade ago, the seminal works of Huang et al. [HCDN08] and Chapman et al. [CJ09] introduced why-not provenance that provides explanations for missing data in the result. Since then, three significant why-not flavors of provenance based explanations have emerged [HDB17]. (i) Instance-based explanations provide data to be inserted in the query’s input. (ii) Query-based explanations reveal operators in the query that need modifications. (iii) Refinement-based explanations yield precise modifications to the query to obtain the missing data. In [HDB17], we provide a comprehensive overview of existing solutions that compute all three flavors of explanations. Here, we summarize the three flavors and distinguish our reparameterization-based explanations and the Breadcrumb algorithm from each flavor’s solutions. Further, we show similarities to non-provenance-based solutions that are related to our reparameterization-based explanations.

2.3.1 Provenance-based explanations for missing data

Here, we discuss the three flavors of provenance-based solutions that compute explanations for missing data.

Instance-based solutions, such as the Missing-Answers algorithm [HH10], the Artemis algorithm [HHT09], the provenance games [KLZ13; LKLG17; LLG20], the Conseil [Her15] algorithm, and Meliou et al.’s algorithm [MGMS10],

compute explanations that suggest inserting tuples in the input. Once tuples are inserted into the input relations as suggested, the output appears in the result. These algorithms are limited to flat relational data. Further, they suggest manipulating the query's input data rather than the query itself, which our reparameterization-based explanations do.

Therefore, our explanations are related closest to query-based why-not provenance solutions since these solutions point at operators in the query that need modification. Existing works in this field [Bel18; BHT14; BHT15; CJ09; DFGH18; DFGH20; Her15] target flat relational data. The solutions [BHT14; BHT15; DFGH18; DFGH20] also support only queries limited to subclasses of the relational algebra extended with aggregation. In contrast, the solutions [Bel18; CJ09] target workflows. They are more flexible than the other solutions regarding query expressiveness. All mentioned solutions are limited to finding explanations based on operators that remove data from the result. Our explanations may yield any operator types in the explanation that have parameters.

Refinement-based provenance solutions, such as the ConQueR [TC10] and the EFQ [TC10] algorithm, do not only point at operators that require modifications. They provide a refined query with operators being rewritten so that the missing data appear in the result. While the Breadcrumb algorithm does not provide rewritten queries, it adopts and extends features from these algorithms. Like the ConQueR algorithm, the Breadcrumb algorithm considers side-effects, i.e. data in the result that appears in the result, but is not needed to explain the missing data. Further, ConQueR considers adding relations to the query if the missing data cannot be produced from the query's input data. Our reparameterization-based explanations extends this idea to alternative attributes in nested data.

In summary, all the mentioned provenance-based solutions that yield explanations for missing data target flat relational data. Consequently, they further lack support for operators that manipulate nested data, such as flattening. Extending these solutions to support nested data is at least as difficult as extending the solutions that compute explanations for existing answers. However, that task is non-trivial, as we have illustrated in Section 2.2.4. The

Breadcrumb algorithm supports nested data and operators manipulating the nested data. It is the first algorithm that computes explanations for missing data over nested data to the best of our knowledge.

Except for the ConQueR algorithm, the introduced algorithms do not take alternative data into account. ConQueR considers other relations for explanations when it cannot generate explanations from the initial input relations. The Breadcrumb algorithm considers individual attributes as alternatives and provides explanations based on these alternatives. Thus, the Breadcrumb algorithm provides explanations that no other provenance-based solutions can provide, even on relational data.

Finally, the mentioned solutions have all been evaluated on datasets with a few thousand to a few million samples. In contrast, our implementation of the Breadcrumb algorithm scales to billions of data samples. To achieve such as scalability to big data, we optimize our implementation.

2.3.2 Non-provenance-based explanations for missing data

Our query-based explanations leverage reparameterizations to find explanations to missing data. Hence, they are not only related to provenance-based solutions that explain missing data. We have identified four non-provenance-based approaches that are related: (i) Query-by-example techniques, (ii) query reverse engineering algorithms (QRE), (iii) interactive query refinement, and (iv) solutions to the empty answer problem.

The Breadcrumb algorithm, in particular, and the query-based explanations, in general, are closely related to query-by-example techniques [DAB16; DG19; Zlo77] and query reverse engineering techniques [Bar19; KLS18; TCP14; TZES17]. The former techniques generate a query from scratch given a set of input-output samples provided by the user. The latter techniques [Bar19; KLS18; TCP14; TZES17] compute a query equivalent to an original, but unknown, query based on the original's query input and output. Like the query-by-example techniques, they generate an entire query. Hence, these techniques resemble algorithms for query-based explanations since both generate explanations based on input data and output data. In

contrast to query-by-example techniques, the algorithms for query-based explanations assume a given query to be erroneous. They do not generate an entire query. Instead, they point to operators in a given query that requires modification.

Further, our explanations for missing data are related to interactive query-refinement techniques and the empty answer problem [MK09; MKZ08; MMR+16]. Query refinements come in two forms: queries can be relaxed to return more results or contracted to return fewer results. The former is commonly used to address the empty answer problem where a query fails to return any result, and the latter to deal with queries that return too many answers. While initially employed for database testing, for instance [MKZ08], recent approaches [MMR+16] offer probabilistic, cost-based means to define different flexible optimization goals that are not limited to mere cardinality constraints. Query refinement describes the relaxation or contraction of a relational query [MK09]. While query relaxation describes query modifications that increase the cardinality of a query's result, query contraction describes query modifications that reduce the cardinality of a query's result. In general, query refinement addresses quantitative constraints on the query result: the rewritten query should return more or less answers. It does not care what these answers are. These constraints are loosely related to the side-effects that Breadcrumb considers for its explanations. Breadcrumb only judges the number of side effects, not their quality. However, the Breadcrumb algorithm distinguishes from query-refinement techniques and empty answer problem because it makes qualitative assumptions on the expected but missing data in the result, which has a specific structure and content.

The techniques mentioned in this section are limited to flat relational data. Unlike the Breadcrumb algorithm, they do not extend to nested data and a query language capable of processing nested data.

2.3.3 Comparison of explanations on the running example

This section illustrates the explanations of the above solutions on the running example from Section 1.2. Neither of the mentioned solutions supports

explanations
operator 2
operator 1

Table 2.5: Query-based explanations for the missing example output based on reparameterizations. The **highlighted explanation** is the only one based on the `address1` attribute used in the original query in Figure 1.1.

nested data, and extending them to nested data is non-trivial. Thus, strictly put, neither of the solutions can compute any of the explanations in Table 1.6. Here, we assume that they support nested data and a query language that is powerful enough to support the running example.

We start with the explanations from Table 1.6 that the Breadcrumb algorithm provides. For convenience, we display them again in Table 2.5. First, we compare our explanations with explanations of related query-based algorithms. These solutions would only yield the **highlighted** explanation. They miss the second explanation, because they require the `flatten` operator to flatten out the with the `address2` attribute instead of the originally referenced `address1` attribute.

The provenance-based solutions that yield refined queries would most likely have provided a query pipeline, as shown in Figure 1.1 with a modified selection condition: `address.year ≥ 2018`. It is an open question, whether these solutions also find explanations that flatten `address2` rather than `address1`, because the `flatten` operator is only applicable on nested data, which the mentioned solutions do not support.

The instance-based solutions would yield an input tuple that complies with the input data, such as the one shown in Table 2.6. The `?` describes placeholders for arbitrary values in the domain of the attribute type. While the values of the `firstname`, the `lastname`, and the `address2` do not matter to generate the missing data in the result, the values in `address1` play a key

firstname	lastname	address1		address2
?	?	city	year	?
		NY	2019	

Table 2.6: Possible instance-based explanation for the missing example output

role. The `address1` attribute has to contain a tuple that has the `city` NY and the `year` 2019. While these solutions also employ provenance to yield the explanations, the explanation from Table 2.6 vastly differs from the ones in Table 2.5 since it targets the input data rather than the query.

The query-by-example techniques could yield a query pipeline that resembles the ones that provenance-based algorithms yield. Here, the provided examples highly impact the resulting pipeline. If the example output was limited to the missing tuple, these techniques would most likely have provided a query pipeline that yields precisely the missing tuple. The query-reverse-engineering techniques are a little more rigid in this sense. They assume that the described output is in the result of an existing but unknown query. Therefore, they require a fully specified output without placeholders, e.g., in the why-not question. Then, they will try to exactly match the output. If only the why-not-tuple is provided, they will generate a query that matches exactly this tuple. If the actual output and the why-not-tuple form the entire output, the query-reverse-engineering techniques will create a query that yields exactly the combined output. However, it may be impossible to create precisely this result, which makes the query-reverse-engineering techniques fail. Both the query-by-example and the query-reverse-engineering techniques are likely to find explanations that build on `address1` as well as `address2`. In that sense, they are closer to our explanations than the provenance-based solutions.

Interactive query-refinement techniques that address the empty answer problem, among other problems, allow the users to make the selection less selective or to modify the flatten operator to flatten `address2` instead of `address1`. Then, the users would find queries that yield the missing answer

more or less by accident. However, the techniques provide no means to the users to request a particular tuple in the result, as Breadcrumb does.

In summary, our query-based explanations that build on reparameterizations combine unique features from provenance-based approaches and non-provenance-based approaches to compute explanations and extends them to nested data.

2.4 Summary

This chapter has provided an overview of the state-of-the-art research on tree-pattern matching and the explanations for existing and missing data. We have found that, in these fields of research, no solutions exist that support nested data and, at the same time, scale to significant amounts of data in big data analytics systems. In the next chapter, we introduce our data and execution model, as well as our system architecture.

DATA MODEL, EXECUTION MODEL, AND ARCHITECTURE

After we have put our work in the context of related research, we introduce the nested data model and execution model as Contribution (1). On purpose, both are similar to the data and execution model supported by big data analytics systems. This similarity ensures that our explanations for existing and missing data, which we define based on the formal models, directly correspond to the data and queries in the systems. The models are published in [DH20b; DLHG21a; DLHG21b]. While the models help to understand the concepts behind our contributions, they shed no light on the implementation. To understand the interplay of the algorithms implemented in the context of this thesis, we provide an overview of our solution's system architecture.

3.1 Data model

Our data model is inspired by the nested relational algebra for bags [GM93] because its data model closely resembles the data format that big data

analytics support. It consists of nested relations, tuples, and primitive types. Nested relations are bags of tuples that share the same type. Tuples have named attributes that are either of primitive types (e.g., booleans or integers), tuple types, or nested relation types. Next, we define the schema and instances of our data. Afterward, we introduce helper functions to access and manipulate the data in this section.

Definition 3.1 (Nested Relation Schema)

Let \mathbb{L} be an infinite set of names. A nested type τ is an element conforming to the context-free grammar below. \mathcal{P} represents a primitive type, \mathcal{R} the nested relation type, and \mathcal{T} the tuple type. The tuple type holds a set of name and type pairs $A_i : \mathcal{A}$. Each name A_i is an element of the name set $A_i \in \mathbb{L}$. The type \mathcal{A} is either of the primitive type \mathcal{P} , the tuple type \mathcal{T} , or the nested relation type \mathcal{R} . Any nested relation type \mathcal{R} is called a nested relation schema. A nested database schema \mathcal{D} is a set of nested relation schemas.

$$\begin{aligned} \mathcal{P} & := INT \mid STR \mid BOOL \mid \dots & \mathcal{R} & := \{\{\mathcal{T}\}\} \\ \mathcal{T} & := \langle A_1 : \mathcal{A}, \dots, A_n : \mathcal{A} \rangle & \mathcal{A} & := \mathcal{P} \mid \mathcal{T} \mid \mathcal{R} \end{aligned}$$

The nested relation schema only defines types over the data. It does not define the actual data values. For that purpose, we introduce instances for the nested types. The structure of an instance I of a type τ is defined as follows.

Definition 3.2 (Nested Relation Instance)

Let \mathbb{I} denote the set of integers, \mathbb{B} either true or false, \mathbb{S} the domain of strings, and \mathbb{D} the domain of datetimes. In all these domains, a special \perp exists denoting the absence of a domain value. The inference rules below recursively define instances I of type τ . The function $\text{type}(A)$ yields the type of any data instance.

$$\begin{array}{c}
\frac{I \in \mathbb{I}}{\text{type}(I) = \text{INT}} \qquad \frac{I \in \mathbb{B}}{\text{type}(I) = \text{BOOL}} \\
\frac{I \in \mathbb{S}}{\text{type}(I) = \text{STR}} \qquad \frac{I \in \mathbb{D}}{\text{type}(I) = \text{DATE}} \\
\frac{\text{type}(I_1) = \tau, \dots, \text{type}(I_n) = \tau}{\text{type}(\{I_1, \dots, I_n\}) = \{\tau\}} \\
\frac{\text{type}(I_1) = \tau_1, \dots, \text{type}(I_n) = \tau_n}{\text{type}(\langle A_1 : I_1, \dots, A_n : I_n \rangle) = \langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle}
\end{array}$$

The inference rules in Definition 3.2 hold conditions in the numerator (top part) and consequence in the denominator (bottom part). For instance, the first rule in Definition 3.2, $\frac{I \in \mathbb{I}}{\text{type}(I) = \text{INT}}$ translates to: “if I is an element of \mathbb{I} , then I ’s type is INT .” Furthermore, the instance definition trivially extend to additional atomic types.

In the following, we will use t, t', t_1, \dots to denote tuples, R, S, T, \dots to denote nested relations, and D, D', \dots to denote nested databases, which are sets of nested relations. As a convention, when R and D are a nested relation and database, then \mathcal{R} and \mathcal{D} denote their types, respectively. In addition to the compact data model representation above, we may also employ an equivalent tree-based representation for visualization purposes. In addition to the type, we define a label function on tuples that returns the set of labels naming the attributes.

Definition 3.3 (Label function)

Given a tuple $t = \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n} \rangle$, the label function $\text{LBL}(t)$ returns the set of attribute labels in t .

$$\text{LBL}(t) := \{A_1, A_2, \dots, A_n\}$$

We overload the label function to also apply on any relation R . Then, $\text{LBL}(R)$ yields the labels of the top-level tuples in the relation: $\text{LBL}(R) = \{\text{LBL}(t)\}$, where $t \in R$. Next, we define the *tuple projection* function that reduces the attributes in a tuple, and the *tuple concatenation* function that merges two input tuples into a single output tuple.

Definition 3.4 (Tuple projection)

Let $t = \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n} \rangle$ be a tuple and L be a set of attribute labels $L \subseteq LBL(t)$. Further, let i be an index over L ($i \in \{1, \dots, l\}$) and j be an index over $LBL(t)$ ($j \in \{1, \dots, n\}$). Then the tuple projection function $t.L$ is defined as:

$$t.L := \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_l : \tau_{A_l} \rangle, \text{ where } \tau_{A_i} = \tau_{A_j} \text{ for } A_i = A_j$$

Definition 3.5 (Tuple concatenation)

Given two tuples $t_A = \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n} \rangle$ and $t_B = \langle B_1 : \tau_{B_1}, B_2 : \tau_{B_2}, \dots, B_m : \tau_{B_m} \rangle$ with disjoint attribute label sets $\{A_1, A_2, \dots, A_n\} \cap \{B_1, B_2, \dots, B_m\} = \emptyset$, the tuple concatenation function $t_A \circ t_B$ concatenates t_A and t_B as follows:

$$\begin{aligned} & t_A \circ t_B \\ &= \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n} \rangle \circ \langle B_1 : \tau_{B_1}, B_2 : \tau_{B_2}, \dots, B_m : \tau_{B_m} \rangle \\ &:= \langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n}, B_1 : \tau_{B_1}, B_2 : \tau_{B_2}, \dots, B_m : \tau_{B_m} \rangle \end{aligned}$$

The above tuple concatenation function is only defined for tuple instances. We overload it to apply to tuple types analogously. Furthermore, we introduce the concatenation function on relation types. Given two relation types $\mathcal{R} = \{\{\tau_{\mathcal{R}}\}\}$ and $\mathcal{S} = \{\{\tau_{\mathcal{S}}\}\}$, the concatenated schema is $\mathcal{R} \circ \mathcal{S} := \{\{\tau_{\mathcal{R}} \circ \tau_{\mathcal{S}}\}\}$.

Furthermore, we introduce the concept of multiplicities. By definition, relations can hold multiple tuples that are indistinguishable by their value and type. Multiplicities indicate the number of these indistinguishable tuples.

Definition 3.6 (Multiplicities)

Given a relation R , $t^n \in R$ denotes that tuple t has the multiplicity n in relation R . In plain, t occurs n times in R . Arithmetic operations on multiplicities of a tuple t indicate that the multiplicity of t amounts to the result of the arithmetic operations. Further, the multiplicity 0 in $t^0 \in R$ indicates the absence of tuple t in relation R . It is equivalent to $t \notin R$. The function $MULT(R, t)$ yields the multiplicity of tuple t in relation R . It returns n for $t^n \in R$.

The multiplicities are relevant in the definition of our execution model. They model that the values and types of tuples in a relation are indistinguishable. However, these tuples may still be distinguishable by a unique identifier, such as the position inside a nested relation. While our execution model does not require these tuples to be distinguishable the tree-pattern matching algorithm, Pebble, and Breadcrumb need distinguishable tuples for distinguishable paths into the nested data. These *paths* allow the algorithms to uniquely identify and access nested attributes, tuples, and relations.

Definition 3.7 (Path)

In the context of tuple t , the following grammar recursively defines the path p :

$$\begin{aligned}
 p &:= t.p' \\
 p' &:= A' \mid A'.p' \\
 A' &:= A \mid A[id]
 \end{aligned}$$

The expression $t.p'$ denotes the projection of tuple t on the sub-path p' . The sub-path p' resolves to either the placeholder A' or the A' associated with a further sub-path $A'.p'$. The placeholder A' resolves to an attribute A of the relation type \mathcal{R} , the tuple type \mathcal{T} , or a primitive type \mathcal{P} . If A' resolves to an attribute A that is of a nested relation type \mathcal{R} , it may also carry an identifier placeholder id , which refers to the element with identifier id in the nested relation A , denoted as $A[id]$. The context of p' is recursively updated to the context A' .

Paths may appear similar to the tuple projection function, since they are both denoted with a “.”. However, the tuple projection function is defined on attribute sets. In contrast, the path is defined on individual attributes.

According to the above definition, the paths only apply to tuples $t \in I$ in relation instances. We overload the path notation to also apply to the schema of the data. A path on the tuple type \mathcal{T} resembles the path on a tuple t . The only difference is regarding the identifier id . Referencing a

nested tuple type \mathcal{T} inside a relation type \mathcal{R} is indicated by keeping the id placeholder of Definition 3.7 in the path. Then, it represents a placeholder for any tuple in the relation instance. Further, we introduce a *root* label to represent the types of top-level tuples in the relation types \mathcal{R} .

We further introduce an identifier function that yields the unique identifiers of tuples and relations. The identifier function $\text{id}(R)$ yields the unique identifier for a nested relation. Similarly, $\text{id}(t)$ provides the identifier of a tuple t that resides in a nested relation. Next, we illustrate our data model and provide an example for the path definition leveraging our running example.

Example 1

The customer data in our example input Table 1.1 conforms to the data model defined in this section. For instance, the tuple t_1 in the table has the following representation in our data model:

$$\left\langle \text{firstname} : \text{Peter}, \text{lastname} : \text{Jones}, \text{address1} : \left\{ \left\{ \langle \text{city} : \text{LA}, \text{year} : 2010 \rangle, \right\} \right\}, \text{address2} : \left\{ \left\{ \langle \text{city} : \text{NY}, \text{year} : 2010 \rangle, \right\} \right\}, \left\{ \left\{ \langle \text{city} : \text{LA}, \text{year} : 2019 \rangle, \right\} \right\}, \left\{ \left\{ \langle \text{city} : \text{LV}, \text{year} : 2017 \rangle, \right\} \right\} \right\rangle$$

The above tuple representation resembles the tabular representation in the input relation Table 1.1. It has the multiplicity $n = 1$, since it occurs exactly once in the input data. Further, t_1 holds the following label set:

$$\text{LBL}(t_1) = \{\text{firstname}, \text{lastname}, \text{address1}, \text{address2}\}$$

Like the other tuples in the input relation, t_1 has the schema $\text{type}(t_1) = \tau_1$:

$$\left\langle \text{firstname} : \text{STR}, \text{lastname} : \text{STR}, \text{address1} : \left\{ \left\{ \langle \text{city} : \text{STR}, \text{year} : \text{INT} \rangle \right\} \right\}, \text{address2} : \left\{ \left\{ \langle \text{city} : \text{STR}, \text{year} : \text{INT} \rangle \right\} \right\} \right\rangle$$

Note that this example does not have a tuple value as an attribute value even though our data model supports it.

We derive the tree-representation of the above tuple and schema straightforwardly. Figure 3.1 shows the above tuple in tree-representation. Analogously to the tuple itself, we can express its schema in the tree-representation, as shown in Figure 3.2. As described above, we name the top-level tuple (type) *root* in both figures.

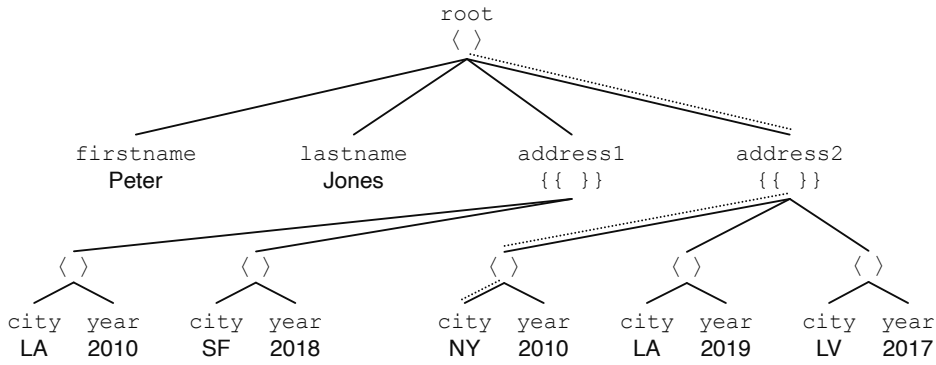


Figure 3.1: First tuple in Table 1.1 represented as a tree. Dotted lines indicate the example path.

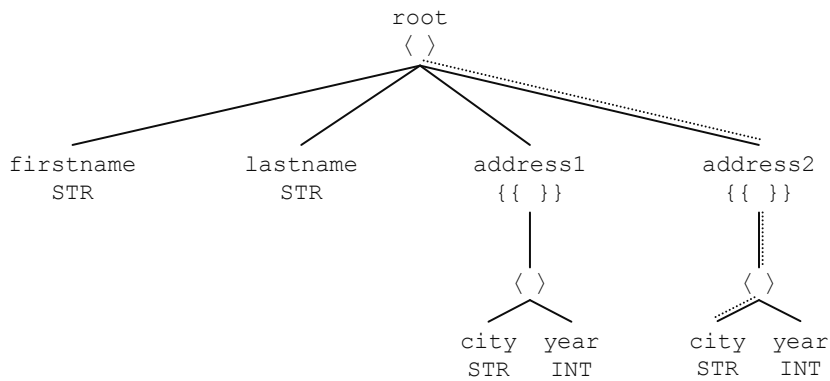


Figure 3.2: Schema of the tuples in Table 1.1 represented as a tree. Dotted lines indicate the example path.

To illustrate our path definition on the tree-representation, we assume that the nested tuple identifiers are 1-indexed from left to right in Figure 3.1. We have added dotted lines to Figure 3.1 and Figure 3.2 to show the example path on the tree-representation. In Figure 3.1, these lines start at the root node and lead to the *city* attribute with the value NY. According to our path definition, the path expressing this route is:

$$root.address2[1].city$$

The path starts at the *root* node and ends at the *city* attribute via the *address2* attribute. Since the *address2* attribute is a nested relation, the path has the nested tuple identifier 1 attached to the attribute reference. Thus,

the path leads precisely to the value NY. Like we apply paths to the data, we apply paths to the schema. In Figure 3.2, we highlight a path from the schema root to the type of the value NY:

root.address2[id].city

This path differs from the previous path on the identifier placeholder [id]. The placeholder indicates that all tuples nested in address2 have the same type, regardless of their identifier value.

Based on the data model defined in this section, we introduce the execution model in the forthcoming section.

3.2 Execution model

The execution model defines the processing semantics of a query Q . These queries process data complying with the data model from Section 3.1. The query is a directed acyclic graph (DAG) of operators, such as filter, flatten or join, which returns a relation after execution. Each of these operators has its own execution semantics. The query execution semantics are motivated by the query execution semantics in big data analytics systems. We want them to closely correspond to obtain meaningful explanations for existing and missing data. In this section, we define the query execution, the operator semantics for each operator in our nested relational algebra for bags (NRAB), and the DAG that associates the operators in a query Q . We start with the query execution.

Definition 3.8 (Query execution)

The execution of a query Q over a database instance D is denoted as $\llbracket Q \rrbracket_D$. The executed query returns a relation of a fixed type denoted as $\text{type}(Q)$ defined inductively over each operator in Q .

We omit the database instance D in $\llbracket Q \rrbracket_D$ and write $\llbracket Q \rrbracket$ instead if D is clear from the context or irrelevant to the discussion. The query Q consists of operators whose semantics we define next.

All operators in our NRAB take one or two input relations and return a single output relation. The two input relations are the n -ary relation R of type $\text{type}(R) = \mathcal{R} := \{\{\langle A_1 : \tau_{A_1}, A_2 : \tau_{A_2}, \dots, A_n : \tau_{A_n} \rangle\}\}$ and the m -ary relation S of type $\text{type}(S) = \mathcal{S} := \{\{\langle B_1 : \tau_{B_1}, B_2 : \tau_{B_2}, \dots, B_m : \tau_{B_m} \rangle\}\}$.

Our algebra supports the operators table access, projection, renaming, selection, union, aggregation, deduplication, and different join, flatten, and nesting operators. It facilitates inner joins, left outer joins, right outer joins, and (full) outer joins. Further, it promotes flatten and nesting for tuples and nested relations since our data model allows nested attributes to have either type. We depict these operators because big data analytics systems natively support these operators. Consequently, we can closely correspond the operators in our algebra to the operators in the systems. That is important to obtain meaningful explanations for existing and missing data in the result.

In [DLHG21a], we show how our operators relate to the operators in a standard nested relational for bags, such as [GM93]. We provide an overview of all supported operators and their semantics in Table 3.1 before we describe each operator below.

Operator	Semantics	Output type $\text{type}(\cdot)$
Table access	$\llbracket R \rrbracket = \{\{t^n \mid t^n \in R\}\}$	\mathcal{R}
Projection	$\llbracket \pi_L(R) \rrbracket = \{\{t^l \mid l = \sum_{t':t'.L=t} \text{MULT}(R, t')\}\}$	$\{\{A_{i_1} : \tau_{i_1}, \dots, A_{i_m} : \tau_{i_m}\}\}$ for $L = \{A_{i_1}, \dots, A_{i_m}\}$
Renaming	$\llbracket \rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R) \rrbracket = \{\{t^l \mid t'^l \in R \wedge t = \langle B_1 : t'.A_1, \dots, B_n : t'.A_n \rangle\}\}$	$\{\{B_1 : \tau_1, \dots, B_n : \tau_n\}\}$ for $\text{type}(A_i) = \tau_i$
Selection	$\llbracket \sigma_\theta(R) \rrbracket = \{\{t^l \mid t^l \in R \wedge t \models \theta\}\}$	\mathcal{R}
Inner Join	$\llbracket R \bowtie_\theta S \rrbracket = \{\{(t \circ t')^{k,l} \mid t^k \in R \wedge t'^l \in S \wedge t \circ t' \models \theta\}\}$	$\mathcal{R} \circ \mathcal{S}$
Left Outer Join	$\llbracket R \bowtie_\theta S \rrbracket = R \bowtie_\theta S \cup \{\{(t \circ t_\perp)^k \mid t^k \in R \wedge t \notin (R \bowtie_\theta S) \wedge t_\perp = \langle B_1 : \perp, \dots, B_m : \perp \rangle\}\}$	$\mathcal{R} \circ \mathcal{S}$
Right Outer Join	$\llbracket R \bowtie_\theta S \rrbracket = R \bowtie_\theta S \cup \{\{(t'_\perp \circ s)^l \mid t'^l \in S \wedge t' \notin (R \bowtie_\theta S) \wedge t'_\perp = \langle A_1 : \perp, \dots, A_n : \perp \rangle\}\}$	$\mathcal{R} \circ \mathcal{S}$
Outer Join	$\llbracket R \bowtie_\theta S \rrbracket = (R \bowtie_\theta S \cup R \bowtie_\theta S) - (R \bowtie_\theta S)$	$\mathcal{R} \circ \mathcal{S}$
Tuple Flatten	$\llbracket F_A^T(R) \rrbracket = \{\{(t \circ t.A)^k \mid t^k \in R\}\}$	$\mathcal{R} \circ \{\{\tau\}\}$
Relation Inner Flatten	$\llbracket F_A^I(R) \rrbracket = \{\{(t \circ u)^{k,l} \mid t^k \in R \wedge u^l \in t.A\}\}$	$\mathcal{R} \circ \tau$
Relation Outer Flatten	$\llbracket F_A^O(R) \rrbracket = F_A^I(R) \cup \{\{(t \circ u_\perp)^k \mid t^k \in R \wedge t \notin (F_A^I(R)) \wedge u_\perp = \langle B_1 : \perp, \dots, B_m : \perp \rangle\}\}$	$\mathcal{R} \circ \tau$
Tuple Nesting	$\llbracket \mathcal{N}_{N \rightarrow C}^T(R) \rrbracket = \{\{(t.G \circ \langle C : t.N \rangle)^k \mid t^k \in R\}\}$	$\{\{\tau_G \circ \langle C : \tau_N \rangle\}\}$, where $G = \text{LBL}(R) - N$
Relation Nesting	$\llbracket \mathcal{N}_{N \rightarrow C}^R(R) \rrbracket = \{\{(t.G \circ ns(R, G, N, C, t))^1 \mid t \in gr(R, G)\}\}$ $gr(R, G) = \{t.G \mid t^n \in R\}$, $ns(R, G, N, C, t) = \langle C : \llbracket \pi_N(\sigma_{t'.G=t.G}(\{\{t' \mid t'^n \in R\}\}) \rrbracket) \rangle$	$\{\{\tau_G \circ \langle C : \{\{\tau_N\}\} \rangle\}\}$, where $G = \text{LBL}(R) - N$
Aggregation	$\llbracket \gamma_{f(A) \rightarrow B}(R) \rrbracket = \{\{(t \circ \langle B : f(t.A) \rangle)^k \mid t^k \in R\}\}$	$\mathcal{R} \circ \{\{B : \text{type}(f(A))\}\}$
Union	$\llbracket R \cup S \rrbracket = \{\{t^{k+l} \mid t^k \in R \wedge t^l \in S\}\}$	\mathcal{R}

Table 3.1: Evaluation semantics and output types for the operators of our NRAB.

Table access. The table access operator reads the relation R without conducting any manipulations on the data. It is denoted as R .

$$\llbracket R \rrbracket := \{\{t^n \mid t^n \in R\}\}$$

$$\text{type}(R) := \mathcal{R}$$

The first line states that the table access operator preserves each tuple's multiplicities in relation R . The second line describes that the type of R is unaffected by the table access.

Projection. Given the input relation R , the projection returns a relation of tuples that hold a specified subset of the input tuples' attributes. Let L be set of labels to be projected on. L is a subset of $L \subseteq \text{LBL}(R)$. Further, let i be an index over L ($i \in \{1, \dots, l\}$) and j be an index over $\text{LBL}(R)$ ($j \in \{1, \dots, n\}$). Then, the projection on a tuple yields the output type $\tau_{out} := \langle A_1 : \tau_{A_1}, \dots, A_l : \tau_{A_l} \rangle$ where $\tau_{A_i} = \tau_{A_j}$ for $A_i = A_j$. Given the constraints, the projection $\pi_L(R)$ of relation R on L is defined as:

$$\llbracket \pi_L(R) \rrbracket := \{\{t^m \mid \text{type}(t) = \tau_{out} \wedge m = \sum_{t': t'.L=t} \text{MULT}(R, t')\}\}$$

$$\text{type}(\pi_L(R)) := \{\{\tau_{out}\}\}$$

The first line ensures that the output relation $\llbracket \pi_L(R) \rrbracket$ has the same number of tuples as the input relation R . Since input tuples t' that are distinguishable in R may become indistinguishable after the projection, the multiplicities of $t \in \llbracket \pi_L(R) \rrbracket$ have to be re-evaluated. The output type of the projection is a bag type of tuple types τ_{out} .

Renaming. Let f be an injective function whose domain is the attribute labels in $\text{LBL}(R)$. We write f as a list of input-output pairs $B_i \leftarrow A_i$, which describe the re-labeling of A_i to B_i . Renaming ρ renames the attributes of

relation R using function f .

$$\begin{aligned} \llbracket \rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R) \rrbracket &:= \\ &\{\{t^l \mid t'^l \in R \wedge t = \langle B_1 : t'.A_1, \dots, B_n : t'.A_n \rangle\}\} \\ \text{type}(\rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R)) &:= \{\{\langle B_1 : \tau_1, \dots, B_n : \tau_n \rangle\}\} \end{aligned}$$

The data types of the renamed attributes remain unaffected by the renaming.

Selection. Let θ be a condition consisting of comparisons between attributes from relation R , constants, and logical connectives. Further, let $t \models \theta$ denote that tuple t fulfills condition θ . The selection $\sigma_\theta(R)$ removes all tuples from R that do not fulfill condition θ .

$$\begin{aligned} \llbracket \sigma_\theta(R) \rrbracket &:= \{\{t^l \mid t^l \in R \wedge t^l \models \theta\}\} \\ \text{type}(\sigma_\theta(R)) &:= \mathcal{R} \end{aligned}$$

Since the selection only removes tuples from the input relation R , it does not influence R 's schema \mathcal{R} .

Join. Let θ be a condition over the attributes of relations R and S . The inner, left outer, right outer, and (full) outer joins are defined as follows:

- **Inner join.** The inner join returns the concatenated tuples from R and S that satisfy the condition θ :

$$\begin{aligned} \llbracket R \bowtie_\theta S \rrbracket &:= \{\{(t \circ t')^{k,l} \mid t^k \in R \wedge t'^l \in S \wedge t \circ t' \models \theta\}\} \\ \text{type}(R \bowtie_\theta S) &= \mathcal{R} \circ \mathcal{S} \end{aligned}$$

- **Left outer join.** The left outer join returns all concatenated tuples of the inner join and tuples from the input relation R that do not join with any tuple in S . To conform to the output schema, the operator extends

them with the attributes from S and assigns them the null value \perp :

$$\begin{aligned} \llbracket R \bowtie_{\theta} S \rrbracket &:= R \bowtie_{\theta} S \cup \{(t_R \circ t_{\perp})^k \mid t^k \in R \\ &\quad \wedge t \notin (R \bowtie_{\theta} S) \wedge t_{\perp} = \langle B_1 : \perp, \dots, B_m : \perp \rangle\} \\ \text{type}(R \bowtie_{\theta} S) &:= \mathcal{R} \circ S \end{aligned}$$

- **Right outer join.** The right outer join returns all concatenated tuples of the inner join and tuples from the input relation S that do not join with any tuple in R . To conform to the output schema, the operator extends them with the attributes from R and assigns them the null value \perp :

$$\begin{aligned} \llbracket R \bowtie_{\theta} S \rrbracket &:= R \bowtie_{\theta} S \cup \{(t'_{\perp} \circ t')^l \mid t'^l \in S \\ &\quad \wedge t' \notin (R \bowtie_{\theta} S) \wedge t'_{\perp} = \langle A_1 : \perp, \dots, A_n : \perp \rangle\} \\ \text{type}(R \bowtie_{\theta} S) &:= \mathcal{R} \circ S \end{aligned}$$

- **Outer join.** The outer join returns all concatenated tuples of the inner join, the tuples from the input relation R that do not join with any tuple in S , and the tuples from the input relation S that do not join with any tuple in R . To conform to the output schema, the operator extends them with the according attributes and assigns them the null value \perp :

$$\begin{aligned} \llbracket R \bowtie_{\theta} S \rrbracket &:= (R \bowtie_{\theta} S \cup R \bowtie_{\theta} S) - (R \bowtie_{\theta} S) \\ \text{type}(R \bowtie_{\theta} S) &:= \mathcal{R} \circ S \end{aligned}$$

The defined join types match commonly available join operators in relational database management systems and big data analytics systems. Thus, we only briefly point out that the types of output relations are the same regardless of the join type.

Flatten. To account for nested tuples and relations, we introduce the flatten operator. It flattens the values of an attribute $A \in \text{LBL}(R)$ into attributes of the top-level tuples, assuming A 's type is either the tuple type or the relation type. We define three different flatten operators to account for the different cases. The tuple flatten operator applies exclusively to tuple types. The inner and outer flatten operators apply to relation types. Conceptually, the inner flatten only returns tuple combination for tuples whose attribute A holds at least one nested tuple. The outer flatten also returns tuple combinations from tuples whose values in A are the empty bag or the null value.

All three flatten operators require that the flattened attribute labels $\text{LBL}(A)$ are distinct from the attribute labels of the top-level tuples $\text{LBL}(R)$, i.e., $\text{LBL}(A) \cap \text{LBL}(R) = \emptyset$.

- **Tuple flatten.** The tuple flatten operator F_A^T unnests an attribute A of tuple type. If A is a tuple type: $\tau = \langle B_1 : \tau'_1, \dots, B_m : \tau'_m \rangle$, then the tuple flatten $F_A^T(R)$ operator semantics are:

$$\begin{aligned} \llbracket F_A^T(R) \rrbracket &:= \{(t \circ t.A)^k \mid t^k \in R\} \\ \text{type}(F_A^T(R)) &:= \mathcal{R} \circ \{\tau\} \end{aligned}$$

- **Inner flatten.** The inner flatten operator F_A^I targets an attribute A that holds a nested relation. If A 's type is a nested relation type: $\tau = \{\{\langle B_1 : \tau'_1, \dots, B_m : \tau'_m \rangle\}\}$, then the inner flatten $F_A^I(R)$ is defined as:

$$\begin{aligned} \llbracket F_A^I(R) \rrbracket &:= \{(t \circ u)^{k.l} \mid t^k \in R \wedge u^l \in t.A\} \\ \text{type}(F_A^I(R)) &:= \mathcal{R} \circ \tau \end{aligned}$$

- **Outer flatten.** Like the inner flatten operator, the outer flatten operator F_A^O unnests an attribute A of nested relation type. In addition to the result of the inner flatten operator, the outer flatten operator yields tuples that have an empty bag \emptyset or the null value \perp in A . It creates a new tuple u_\perp of type $\text{type}(u_\perp) = \text{type}(A)$ to concatenate it with these tuples. If A 's type is a nested relation type: $\tau = \{\{\langle B_1 : \tau'_1, \dots, B_m : \tau'_m \rangle\}\}$,

then the outer flatten $F_A^O(R)$ is defined as:

$$\begin{aligned} \llbracket F_A^O(R) \rrbracket &:= F_A^I(R) \cup \{(t \circ u_\perp)^k \mid t^k \in R \\ &\quad \wedge (t.A = \emptyset \vee t.A = \perp) \wedge u_\perp = \langle B_1 : \perp, \dots, B_m : \perp \rangle\} \\ \text{type}(F_A^O(R)) &:= \mathcal{R} \circ \tau \end{aligned}$$

Nesting. Our algebra offers two nesting operators: the tuple nesting operator to create nested tuples and the relation nesting operator to create nested relations. Before we introduce the operators, we define the nesting attributes N and the grouping attributes G over the input relation R . Let $N \subseteq \text{LBL}(R)$ be the nesting attributes, then $G = \text{LBL}(R) - N$ is the set of grouping attributes. Further, let t be a tuple in R , then the projections $t.N$ and $t.G$, yield tuples of type τ_N and τ_G , respectively.

- **Tuple nesting.** The tuple nesting operator $\mathcal{N}_{N \rightarrow C}^T$ constructs a tuple from the nesting attributes N and nests this tuple into a new tuple-typed attribute C . It concatenates $t.G$ with the attributes obtained from a tuple-projection $t.N$ for the result.

$$\begin{aligned} \llbracket \mathcal{N}_{N \rightarrow C}^T(R) \rrbracket &:= \{(t.G \circ \langle C : t.N \rangle)^k \mid t^k \in R\} \\ \text{type}(\mathcal{N}_{N \rightarrow C}^T(R)) &:= \{\tau_G \circ \langle C : \tau_N \rangle\} \end{aligned}$$

- **Relation nesting.** The relation nesting $\mathcal{N}_{N \rightarrow C}^R$ operator groups the input tuples based on the grouping attributes G . For each group, it yields a tuple with the attributes in G plus a new attribute C of relation type that contains all tuples in the group. Before the operator nests these tuples, it applies a projection on the tuples on the nesting attributes N .

$$\begin{aligned}
\llbracket \mathcal{N}_{N \rightarrow C}^R(R) \rrbracket &:= \{(t.G \circ ns(R, G, N, C, t))^1 \mid t \in gr(R, G)\} \\
gr(R, G) &:= \{t.G \mid t^n \in R\} \\
ns(R, G, N, C, t) &:= \langle C : \llbracket \pi_N(\sigma_{t'.G=t.G}(\{t' \mid t'^n \in R\})) \rrbracket \rangle \\
type(\mathcal{N}_{N \rightarrow C}^R(R)) &= \{\tau_G \circ \langle C : \{\tau_N\} \rangle\}
\end{aligned}$$

The definition of the relation nesting operator utilizes two helper functions $gr(R, G)$ and $ns(R, G, N, C, t)$. While the former function creates the groups, the latter function creates the nested relations for the new attribute C .

Aggregation. The aggregation operator $\gamma_{f(A) \rightarrow B}$ aggregates the values in a nested relation into a new value based on the aggregation function f . We are given the aggregation function f of type $\{\tau\} \rightarrow \tau_{out}$, where τ_{out} represents a primitive type \mathcal{P} . For each tuple t in the input R the aggregation operator applies f to attribute A 's value, which has to be of relation type $type(A) = \{\tau\}$. The function returns a value of τ_{out} , which the operator associates with the attribute label B and adds to the output tuple.

$$\begin{aligned}
\llbracket \gamma_{f(A) \rightarrow B}(R) \rrbracket &:= \{(t \circ \langle B : f(t.A) \rangle)^k \mid t^k \in R\} \\
type(\gamma_{f(A) \rightarrow B}(R)) &:= \mathcal{R} \circ \langle B : \tau_{out} \rangle
\end{aligned}$$

Union. The union operator merges the input relations R and S , which have the same type $type(R) = type(S)$ into a single output relation. Further, $t^0 \in R$ is true by convention if the tuple t is not part of relation R .

$$\begin{aligned}
\llbracket R \cup S \rrbracket &:= \{t^{k+l} \mid t^k \in R \wedge t^l \in S\} \\
type(R \cup S) &:= \mathcal{R}
\end{aligned}$$

A query Q is a composition of the operators defined above. Each operator O in the query Q is identifiable by a unique identifier $id(O)$. To illustrate

how to compose a query from the operators, we apply the algebra on our example query shown in Figure 1.1.

Example 2

Given the input data from Table 1.1, which we call $R_{example}$ from here on, the example query $Q_{example}$ illustrated in Figure 1.1 has the following representation in our algebra:

$$\mathcal{N}_{firstname \rightarrow nList}^R \left(\pi_{\{firstname, city\}} \left(\sigma_{year \geq 2019} \left(F_{address1}^I (R_{example}) \right) \right) \right)$$

The query $Q_{example}$ has four operators, like the pipeline in Figure 1.1. It first flattens the *address1* attribute $F_{address1}^I$, before it applies the filter $\sigma_{year \geq 2019}$ on the *year*. Then, the query retains only the *firstname* and *city* by applying a the projection $\pi_{\{firstname, city\}}$. Finally, it nests the firstnames into a nested relation named *nList* with the nesting operator $\mathcal{N}_{firstname \rightarrow nList}^R$. When the query $Q_{example}$ is executed, $\llbracket Q_{example} \rrbracket$ yields the relation displayed in Table 1.2.

The above data and execution model address this thesis' Contribution (1). We leverage them to formally introduce the tree-pattern matching algorithm (Contribution (2)), the explanations for existing data (Contribution (3)), and the explanation for missing data (Contribution (3)) in the following chapters. They are system-independent, i.e., they do not designed with a particular big data analytics system in mind. Given that the overall research question of this thesis focusses on big data analytics systems, it is nonetheless crucial to devise an architecture that exploits common features these systems. Hence, in the next section, we propose a system architecture that illustrates the interplay of our algorithms with each other and the big data analytics system.

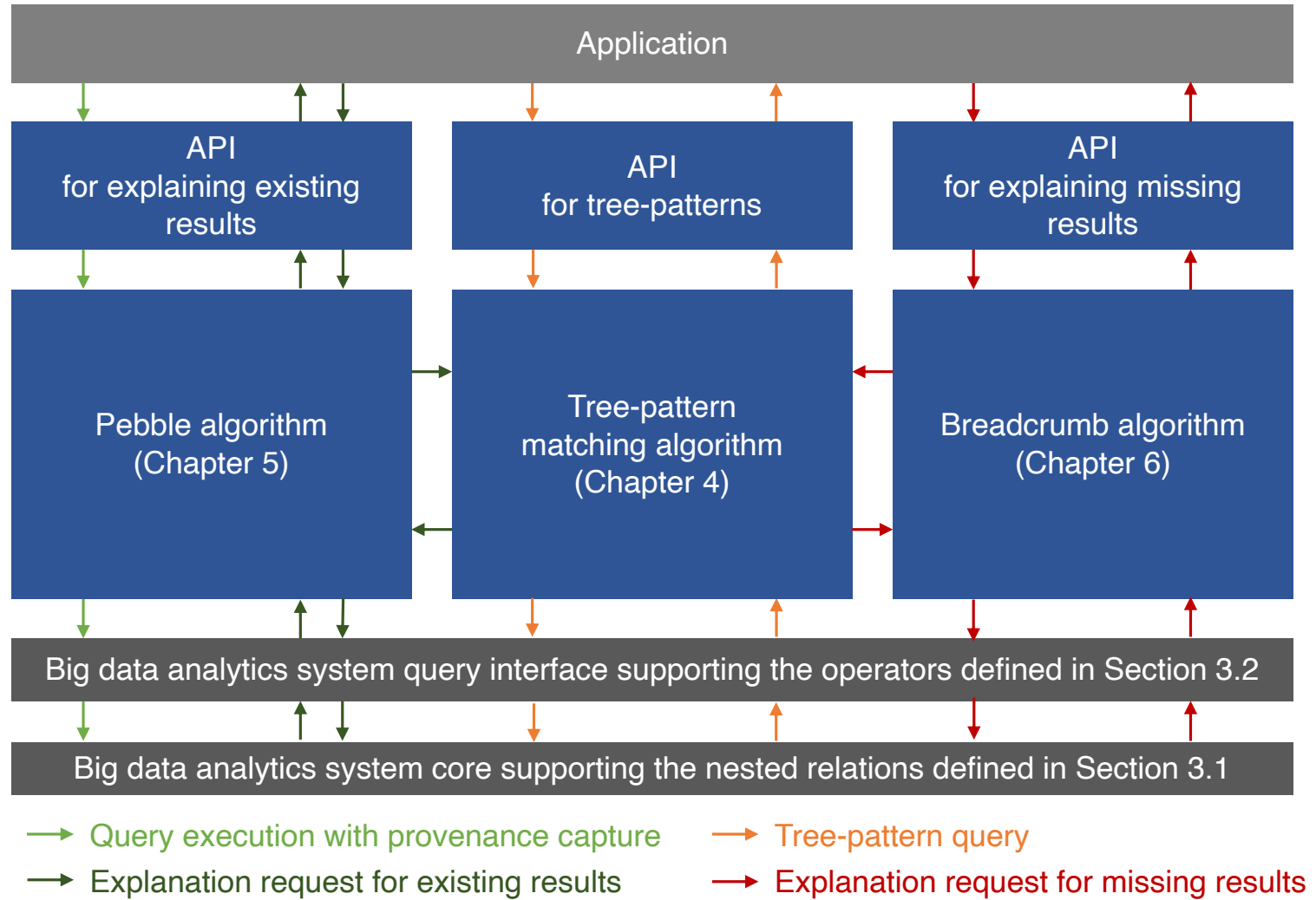


Figure 3.3: System architecture. Spark modules and applications are grey. Our system's modules are blue. The arrows indicate the execution interactions of the modules.

3.3 Architecture

The tree-pattern matching algorithm, the Pebble algorithm that computes explanations for existing data, and the Breadcrumb algorithm that computes explanations for missing data are designed for big data analytics systems. To understand how they interact with each other and with the analytics system, we introduce our system architecture in this section.

Figure 3.3 provides an overview of the system architecture. In the center, it shows the three algorithms in blue. The applications on the top (light grey) access the algorithms via an API (blue). Furthermore, the algorithms access the big data analytics system on the bottom (dark grey) via an interface that supports the operators defined in 3.2. Internally, the analytics system processes data complying with the data model defined in 3.1. The arrows in Figure 3.3 describe the interactions between the modules, the application, and the big data analytics system. In the following, we describe the blue algorithm modules along with the corresponding interactions.

We start with the tree-pattern matching module in the center of Figure 3.3 that addresses Contribution (2). It implements a tree-pattern matching algorithm that scales to large datasets in big data analytics systems because it avoids global state and large amounts of intermediate results. As the orange arrows show, applications call the tree-pattern matching algorithm in the corresponding module via a dedicated API. The API passes the tree-pattern to the tree-pattern matching algorithm that we describe in detail in Chapter 4. The algorithm has two major steps. The first step - the schema matching step - computes schema-matches from the tree-pattern and the relation's schema without accessing the data values. The second step - the data matching step - generates a big data analytics query based on these schema-matches. By executing the query, it obtains the matching data from the system. The algorithm returns the matching data to the application via the API.

To query explanations for existing data, the applications leverage the Pebble algorithm module shown in blue on the left in Figure 3.3. It addresses Contribution (3) since it provides fine-grained explanations based

on structural provenance. We describe the details of the Pebble algorithm in Chapter 5. Here, we focus on Pebble’s interaction with the applications, the big data analytics system, and the tree-pattern matching module. Recall that Pebble eagerly captures the provenance. Therefore, the application has to activate Pebble before query execution. The light-green arrows show the activation and capture process. The application requests the Pebble module to capture the provenance for an analytical query. Then, the Pebble module triggers the execution of the query on the big data analytics system via the query interface. During the execution, Pebble collects the structural provenance and stores it in the big data analytics system. It does not return any explanations, so far. Hence, the light-green arrows do not return to the application but end in the big data analytics system.

To obtain explanations another Pebble API call is necessary as indicated by the dark-green arrows in Figure 3.3. This call contains the previously executed query and a tree-pattern that describes the existing data of interest. Once the API module receives the call, it checks that the provenance has been collected for the given query. It returns an error if the check is not successful. If the check succeeds, the API forwards the request to the Pebble module. This module first matches the input tree-pattern onto the query’s result data. For that purpose, it leverages the tree-pattern matching module. That module yields all data in the result that match the tree-pattern. Given the matching data, the Pebble module moves backward in the query operator by operator. It leverages the previously captured provenance to undo manipulations on the data structure and dissolves data dependencies. Once it reaches the input, it maps the traced data structures and dependencies onto the input data to obtain the explanation. It returns the explanation to the application via the API.

While the Pebble module computes explanations for existing data, the Breadcrumb module calculates explanations for missing data. The Breadcrumb algorithm addresses our research Contribution (4) since it yields query-based explanations for missing data leveraging the novel concept of reparameterizations. We describe it in detail in Chapter 6. Here, we focus on Breadcrumb’s interactions with the tree-pattern matching module and

the big data analytics system. The red arrows in Figure 3.3 describe the control flow. To obtain an explanation, the application provides a why-not question to the Breadcrumb API. This question references the input data, the query, and a tree-pattern that describes the missing data. The API forwards the question to the Breadcrumb module. That module leverages the schema matching step of the tree-pattern matching algorithm to identify attributes that match the tree-pattern's nodes. Then, it traces these attributes back to the query's input relations. To identify data that potentially contribute to the missing data described in the tree-pattern, Breadcrumb applies the data matching step of the tree-pattern matching algorithm with the backtraced attributes. Then, Breadcrumb re-executes the query and forward traces the data through the query. During the query re-execution, Breadcrumb adds annotations to the processed data. After the query execution, Breadcrumb computes the approximated explanations based on the annotations and returns them to the application via the API.

3.4 Summary

In this chapter, we have defined a nested data model and an execution model that closely resemble the data and execution model of big data analytics systems. These models form the basis for the following contributions. We have further provided an architecture overview that illustrates the interplay of the algorithms developed and implemented in the context of this work.

In the following, we dedicate one chapter to each of the three algorithms to describe the internals the modules' internals in detail. Since Pebble and Breadcrumb rely on the tree-pattern matching module, we start with the tree-pattern matching in the next chapter.

TREE-PATTERN MATCHING

Today's big data analytics systems provide rudimentary means to address individual nested data values. They lack sophisticated means to address arbitrary combinations of them. Tree-patterns allow us to declaratively address combinations of nested data related by their structure, constrained by their value, or restricted by their number of occurrences. They are not only necessary to query explanations but also convenient to query nested data directly. However, big data analytics systems do not support them because state-of-the-art tree-pattern matching algorithms heavily rely on global state. That inhibits distributed computation. Thus, we devise a novel tree-pattern matching algorithm for big data analytics systems that scales to large dataset sizes. It utilizes the fact that big data systems process nested relations of data that share the same schema. In the first step, it matches the tree-pattern exclusively on the schema. In the second step, it accesses the actual data to obtain the tree-pattern matches. The algorithm can parallelize and distribute the execution of the second step to achieve scalability. As illustrated in Section 3.3, it utilizes the big data analytics system for that purpose.

This chapter describes Contribution (2) in detail and is based on our publication on distributed tree-pattern matching [DH20a]. In the beginning, we introduce our tree-pattern syntax and formally describe when a tree-pattern matches data corresponding to our data model in Section 3.1. Then, we introduce the distributed tree-pattern matching algorithm. We illustrate both in our running example. We conclude the chapter with a discussion on the algorithm’s complexity.

4.1 Tree-pattern syntax

In Figure 1.2, we have informally introduced two example tree-patterns that describe the highlighted data in Table 1.3 and Table 1.4. Furthermore, we have informally described the supported syntax elements in Figure 1.2c. We define value constraints on nodes, cardinality constraints on nodes, and the tree-pattern nodes first. Then, we define structural constraints and tree-pattern edges. Eventually, we introduce tree-patterns.

The constraints are necessary to precisely address arbitrary combinations of nested data. Informally described, value constraints make a tree-pattern node match an attribute only if the node name matches the attribute name, and the value constraint matches the attribute value. They only apply to primitive types.

Definition 4.1 (Value constraint)

A value constraint $vc = \langle comp, const \rangle$ with respect to an attribute A of primitive type \mathcal{P}_A is a tuple holding a comparison operator $comp \in \{=, \neq, >, \geq, <, \leq, contains\}$ and a constant value $const$ in the domain of \mathcal{P}_A , denoted as $\tau(const) = \mathcal{P}_A$.

Since value constraints only apply to primitive types, they must be leaf nodes in our tree-pattern, i.e., they must not have any descendants. In contrast, cardinality constraints do not apply to leaf nodes. If a node holds a cardinality constraint, it only matches attributes of relation type \mathcal{R} . The constraint defines an upper and lower bound that constrains the occurrence of the node’s subtree in the data.

Definition 4.2 (Cardinality constraint)

A cardinality constraint with respect to an attribute A of relation type \mathcal{R} is a tuple $cc = \langle min, max \rangle$ with $min, max \in \mathbb{N}^+ \cup \infty$ defining the minimum and the maximum number of occurrences of tuples in the instance of A .

The default lower bound is $min = 1$ and the default upper bound is $max = \infty$. Hence, a node without an explicit cardinality constraint, may still match an attribute of relation type \mathcal{R} . Given the value and cardinality constraints, we define the tree-pattern nodes. Each tree-pattern node has a label to match an attribute label in the data. Furthermore, it has an attribute for a value and a cardinality constraint. It additionally has a boolean value that describes whether the value on which the tree-pattern node matches becomes part of the tree-pattern matching output. The tree-pattern node is the following 4-tuple.

Definition 4.3 (Tree-pattern node)

A tree-pattern node n is a 4-tuple $n = \langle A, vc, cc, on \rangle$, where $A \in \mathbb{L} \cup \{\{root\}\}$ is an attribute name, vc either \perp or a value constraint, cc is a cardinality constraint, and the boolean value on is *true* if it is an output node and *false* otherwise.

As mentioned in the running example Section 1.2 and in the data model Section 3.1, we introduce a *root* label to refer to top-level tuples in the input relation. Edges connect the tree-pattern nodes to form a fully specified tree-pattern. These edges have either of two types, which define structural constraints over the data.

Definition 4.4 (Structural constraint)

A structural constraint with respect to two tree-pattern nodes n_1 and n_2 is a boolean value $sc = \{true|false\}$ that encodes a parent-child relationship (PC) between n_1 and n_2 when *true* or an ancestor-descendant relationship (AD) when *false*.

Given the structural constraints, a tree-pattern edge is:

Definition 4.5 (Tree-pattern edge)

A tree-pattern edge is a triple $e = \langle n_1, n_2, sc \rangle$, where $n_1 \neq n_2$ are tree-pattern nodes and sc is a structural constraint with respect to nodes n_1 and n_2 .

The tree-pattern is a tree whose edges connect the tree-pattern nodes. It roots in a dedicated node that is the only node without a parent.

Definition 4.6 (Tree-pattern)

A tree-pattern is a triple $T = \langle r, N, E \rangle$. Here, r is a tree-pattern node, N is a set of tree-pattern nodes (including r), and E is a set of tree-pattern edges. N and E form a single tree rooted at r .

To illustrate our tree-pattern definition, we demonstrate it on the running example.

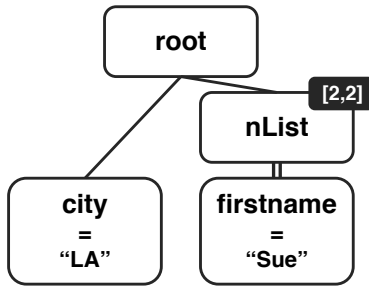


Figure 4.1: Example tree-pattern T (Figure 1.2a)

city		nList	
101	LA	firstname	
		711	Sue
		712	Sue
		713	Tom

Table 4.1: Unexpected data that is **existing** in the example output (Table 1.3)

Example 3

Recall that our example tree-pattern describes the existing, but unexpected data in our running example. For convenience, we show the tree-pattern again in Figure 4.1 and the unexpected output data again in Table 4.1.

Now, we express the tree-pattern in our formal tree-pattern syntax. It has four nodes, of which two have a value constraint, and one has a cardinality constraint. Two out of three edges describe a PC-relationship. We first define the set of tree-pattern nodes $N_{existing}$, starting with the root node:

$$N_{existing} = \left\{ \begin{array}{l} n_{root} = \langle root, \perp, \langle 1, \infty \rangle, false \rangle, \\ n_{city} = \langle city, \langle =, LA \rangle, \langle 1, \infty \rangle, false \rangle, \\ n_{nList} = \langle nList, \perp, \langle 1, 2 \rangle, false \rangle, \\ n_{firstname} = \langle firstname, \langle =, Sue \rangle, \langle 1, \infty \rangle, false \rangle \end{array} \right\}$$

Like the tree-pattern in Figure 4.1, the set of nodes $N_{existing}$ has four nodes. The two tuples $\langle =, LA \rangle$ and $\langle =, Sue \rangle$ represent the value constraints in the *city* and *firstname* node, respectively. Further, the tuple $\langle 2, 2 \rangle$ describes the cardinality constraint on the *nList* node. To connect these nodes, we define the set of edges $E_{existing}$ next.

$$E_{existing} = \left\{ \begin{array}{l} \langle n_{root}, n_{city}, true \rangle, \\ \langle n_{root}, n_{nList}, true \rangle, \\ \langle n_{nList}, n_{firstname}, false \rangle \end{array} \right\}$$

The top two edges define PC relationships between the root node and the *city* node, and the root node and the *nList* node, respectively. The lower edge describes an AD relationship between the *nList* node and the *firstname* node. Given the set of nodes $N_{existing}$ and the set of edges $E_{existing}$, the tree-pattern $T_{existing}$ is:

$$T_{existing} = \langle n_{root}, N_{existing}, E_{existing} \rangle$$

Unlike our tree-pattern $T_{existing}$, the built-in means of state-of-the-art big data analytics systems cannot directly address and associate the two nested firstnames *Sue*. The following section describes the tree-pattern matching algorithm, which identifies the highlighted data in Table 4.1, given $T_{existing}$.

In this section, we have defined the tree-pattern syntax. In the following section, we formally define when a tree-pattern matches the data.

4.2 Tree-pattern matching definition

Once the tree-pattern is defined, it has to be matched on the data in our nested relations. Here, we introduce a novel matching definition for matching tree-patterns on nested relations corresponding to our data model in Section 3.1. The definition distinguishes from previous definitions, since it explicitly exploits the fact that data in our nested relations share the same schema. Previous approaches have mainly targeted semi-structured XML data sources, for which the schema property generally does not hold. Based on our matching definition we can devise our novel, distributed tree-pattern matching algorithm in the next section.

Given a Tree-pattern T and a relation R , tree-pattern matching identifies all data in R that have counterparts to all nodes N and all edges E in T . Further, the data must fulfill all value, cardinality, and structural constraints that the pattern defines.

According to our data model in Section 3.1, all tuples in a nested instance comply with the same schema. Our tree-pattern matching algorithm utilizes this property to divide the matching into a data-independent schema-matching and a data-dependent data-matching phase. During schema-matching, the matching algorithm checks for matches in (i) the node names in the tree-pattern and (ii) compliance to all structural constraints. During data-matching, it further checks cardinality and value constraints. In the following, we formally define the schema-matching and data-matching before we introduce our two-phase tree-pattern matching algorithm. The schema matching is defined as follows.

The schema-matching finds all possible trees that only have parent-child edges, correspond to the nested relation's schema and satisfy all structural conditions in the tree-pattern.

Definition 4.7 (Schema-matching)

Given $T = \langle r, N, E \rangle$ and the schema \mathcal{R} of a nested relation R , a schema match M fulfills the following constraints over \mathcal{R} :

1. All nodes $n \in N$ match an attribute label $A_R \in \mathcal{R}$, denoted $M(n) = A_R$:
 $\forall n = \langle A, vc, cc, on \rangle \in N, \exists A_R \in \mathcal{R} \models A = A_R$
2. All PC relationships $e_{PC} \in E$ match in \mathcal{R} , denoted $M(e_{PC}) = \langle A_P, A_C \rangle$:
 $\forall e_{PC} = \langle n_P, n_C, true \rangle \in E, \exists A_P, A_C \in \mathcal{R} \models M(n_P) = A_P \wedge M(n_C) = A_C \wedge (\text{path}(A_C) = \text{path}(A_P).A_C \vee \text{path}(A_C) = \text{path}(A_P)[id].A_C)$
3. All AD relationships $e_{AD} \in E$ match in \mathcal{R} , denoted $M(e_{AD}) = \langle A_A, A_D \rangle$:
 $\forall e_{AD} = \langle n_A, n_D, false \rangle \in E, \exists A_A, A_D \in \mathcal{R} \models M(n_A) = A_A \wedge M(n_D) = A_D \wedge \text{path}(A_A) \text{ is a prefix of } \text{path}(A_D)$

Each match M maps T to a tree with nodes M_N and edges M_E , i.e., $M(T) = \langle M_N, M_E \rangle$ where $M_N = \bigcup_{n \in N} M(n)$ and $M_E = \bigcup_{e \in E} M(e)$. In this tree, nodes are uniquely identified by their path and edges by the according path pairs. All value constraints vc , cardinality constraints cc , and output flags on are transferred from the nodes in T to their matching nodes in M . Furthermore, the tree-pattern T may match multiple times onto the schema \mathcal{R} . The schema-matching \mathcal{M} is the set of all possible schema-matches of T in \mathcal{R} .

The schema-matches in \mathcal{M} serve as blueprints for the full tree-pattern matches over the input relation R of type \mathcal{R} . They guarantee that all required attribute labels are present and all structural constraints are satisfied. Therefore, in the data-matching step, the algorithm only checks the cardinality constraints and value constraints for each match $M \in \mathcal{M}$ on each tuple $t \in R$.

Definition 4.8 (Data-matching)

Given a schema match M and a tuple $t \in R$, t is a data match of M , if:

1. all value constraints hold:
 $\forall n \in M_N, vc_n \neq \perp \models \text{the value of } \text{path}(n)^t \text{ satisfies } vc$
2. all cardinality constraints hold:
 $\forall n \in M_N : cc_n \neq \perp \models \text{subtree of } M \text{ rooted at } n \text{ matches tuples in the nested relation at } \text{path}(n)^t \text{ at least } \text{min}_{cc} \text{ times and at most } \text{max}_{cc} \text{ times}$

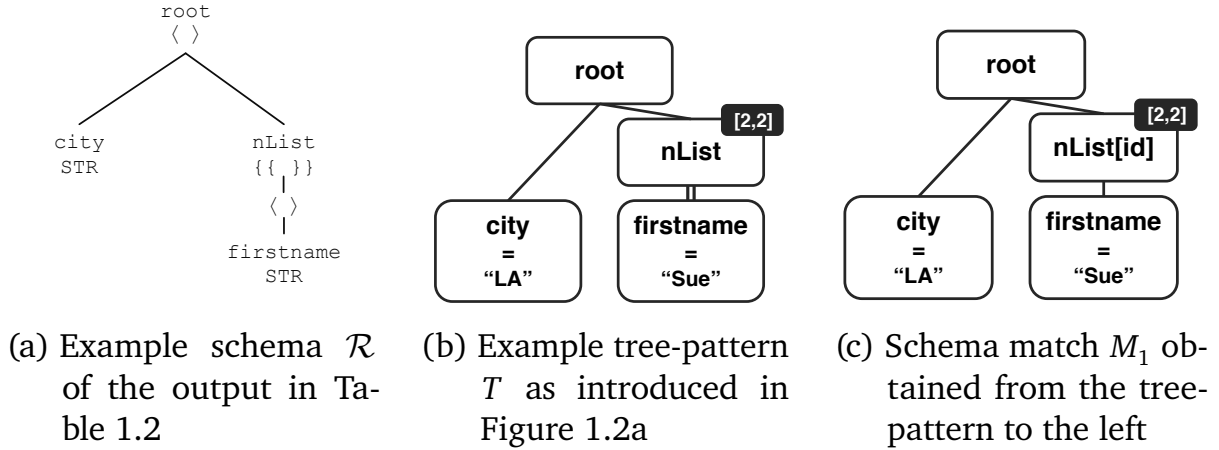


Figure 4.2: Schema matching example

A tuple $t \in R$ may match multiple schema-matches $M \in \mathcal{M}$. We illustrate the schema-matching and data-matching on our running example, before we discuss the algorithm.

Example 4

Recall that we have shown the example data and tree-pattern again at the beginning of this chapter in Table 4.1 and Figure 4.1, respectively. For convenience, we show the schema \mathcal{R} of the example output in Figure 4.2a, the example tree-pattern in Figure 4.2b. Matching the tree-pattern in Figure 4.2b onto the schema \mathcal{R} in Table 4.1 yields the schema match M_1 in Figure 4.2c. As described above, each node in the tree-pattern has a counterpart in the schema match M_1 . Further, the schema match does not have any ancestor-descendant (AD) edges anymore, unlike the tree-pattern. It has an AD edge between the $nList$ node and the $firstname$ node. Furthermore, all value and cardinality constraints are transferred from the tree-pattern to the according nodes in M_1 .

The data matching applies the schema match M_1 on each tuple in the result R in Table 4.1 to validate the value and cardinality constraints. The only tuple t in R fulfills both value constraints since the $city$ attribute holds the value “LA” and the nested relation $nList$ holds tuples whose attribute $firstname$ matches “Sue”. Since the relation has exactly two occurrences of “Sue”, the tree-pattern T matches the tuple t .

We refer the interested reader to [DH20a] for a more complex example with multiple matches.

4.3 Tree-pattern matching algorithm

Given the tree-pattern T and the dataset instance R of schema \mathcal{R} , we propose a novel two-phase algorithm to match T onto R . It is designed for distributed execution on big data analytics systems since it leverages the schema properties of the nested relations and avoids global state. The algorithm has two phases that correspond to the definitions of the schema-matching and the data-matching provided in Section 4.2. Algorithm 1 provides an overview of the two phases.

Algorithm 1: $tpm(T, R)$

Input: Tree-pattern T , relation R of type \mathcal{R}

Output: Relation R' of type \mathcal{R}'

- 1 $\mathcal{M} \leftarrow matchSchema(T, \mathcal{R})$
 - 2 $R' \leftarrow matchData(\mathcal{M}, R)$
 - 3 **return** R'
-

The *matchSchema* function computes the schema matches \mathcal{M} as defined in Definition 4.7 based on the tree-pattern T and the relation R 's schema \mathcal{R} . It does not access the data in R . The algorithm exclusively accesses R when calling the *matchData* function. Given the schema matches \mathcal{M} and the relation R , this function yields the output relation R' with the matching tuples. The separation of the schema-matching and data-matching phases allows for distributed tree-pattern matching, because the algorithm can apply the data matching phase in parallel on the tuples in R . That is imperative when processing data in big data analytics systems. In the following, we describe the two phases of the algorithm in detail.

Algorithm 2: *matchSchema*(T, \mathcal{R})

Input: Tree-pattern T , schema \mathcal{R} **Output:** SchemaMatches \mathcal{M}

```
1  $\mathcal{M} \leftarrow \emptyset$ 
2  $\mathcal{R}_D \leftarrow \text{deweyID}(\mathcal{R})$ 
3  $I_{\mathcal{R}} \leftarrow \text{deweyIdx}(\mathcal{R}_D)$ 
4  $RC \leftarrow I_{\mathcal{R}}(T.r.name)$ 
5 for  $rc \in RC$  do
6    $C \leftarrow \emptyset$ 
7   for  $l \in \text{leaves}(T)$  do
8      $C \leftarrow C \cup \langle \text{node} : l, \text{candidates} : \{\{lc \mid lc \in$ 
9        $I_{\mathcal{R}}(l.name) \wedge \text{prefix}(rc, lc) = \text{true}\}\} \rangle$ 
10    for  $c \in \{\{\langle cc_1.\text{node} : lc_1, \dots, cc_{|C|}.\text{node} : lc_{|C|} \rangle \mid lc_i \in$ 
11       $cc_i.\text{candidates} \wedge cc_i \in \text{enum}(C)\}\}$  do
12       $M_c \leftarrow \text{merge}(rc, c)$ 
13      if  $\text{validate}(M_c) = \text{true}$  then
14         $\mathcal{M} \leftarrow \mathcal{M} \cup \{\{\text{transferConstraints}(M_c)\}\}$ 
15 return  $\mathcal{M}$ 
```

4.3.1 Schema matching

Algorithm 2 computes the set of schema matches \mathcal{M} . During the initialization phase (ll. 1-4), the algorithm labels each attribute in \mathcal{R} with a unique DeweyID to obtain the annotated schema \mathcal{R}_D . The DeweyID [LLCC05] encodes the nesting and sibling relationship of attributes. Based on these ids, the algorithm creates an index $I_{\mathcal{R}}$, that maps each unique node label to a collection of DeweyIDs. Next, it retrieves the DeweyIDs of all root candidates RC from the index. All identifiers in RC match the root node r of T (l. 4).

While iterating over the root candidates (ll. 5-12), the algorithm computes the schema matches \mathcal{M} for each root candidate $rc \in RC$. First, it associates each leaf node l in the tree-pattern T with a set of candidate attributes of \mathcal{R}_D in a map C . It identifies the attributes with the DeweyID (ll. 6-8). The candidate matches to each leaf node l are the nodes that match l 's label

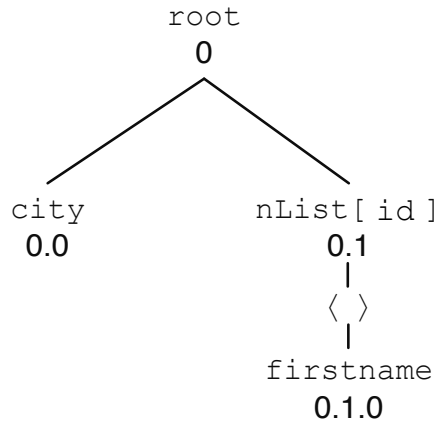


Figure 4.3: Example DeweyIDs applied on the schema in Figure 4.2a

and that are descendants of the root candidate ($prefix(rc, lc) = true$). The algorithm retrieves them from index $I_{\mathcal{R}}$ ($lc \in I_{\mathcal{R}}(l.name)$).

In lines 9-12, the algorithm computes the schema matches from C . It computes the cross product of the DeweyIDs stored in each leaf label's set in C (l. 9). Then, for each combination c comprising $|C|$ DeweyIDs, the algorithm merges the root to leaf paths into a schema match candidate M_c using the DeweyIDs (l. 10). Next, it validates that M_c fulfills the tree-pattern T 's structural constraints. When the validation is successful, it transfers the value and cardinality constraints from nodes in T to the corresponding nodes in M_c and adds M_c to \mathcal{M} .

Example 5

When the algorithm computes the schema matches from the tree-pattern in Figure 4.2b and the schema in Figure 4.2a, it assigns each attribute in the schema a unique DeweyID, as shown in Figure 4.3. For instance, the *nList* attribute has the label 0.1. It is a child of the *root* with label 0 and sibling of the *city* with label 0.0. Further, it is the parent to *firstname* with the label 0.1.0. The unlabeled tuples in nested relations indicated by the tuple-brackets between the *nList* and the *firstname* do not receive a DeweyID, since they do not have a label.

From the schema with DeweyIDs, the algorithm computes the index $I_{\mathcal{R}}$.

$$I_{\mathcal{R}} = \left\{ \begin{array}{l} \langle root : \{0\} \rangle, \\ \langle city : \{0.0\} \rangle, \\ \langle nList : \{0.1\} \rangle, \\ \langle firstname : \{0.1.0\} \rangle \end{array} \right\}$$

In this example, each label occurs precisely once. Thus, each entry in $I_{\mathcal{R}}$ contains one DeweyID. In general, the entries can contain multiple identifiers. The algorithm obtains the root candidate set $RC = \{\{0\}\}$ from $I_{\mathcal{R}}$. It contains only the root's DeweyID 0. Then, the algorithm computes the following leaf candidate mapping C :

$$C = \left\{ \begin{array}{l} \langle city : \{0.0\} \rangle, \\ \langle firstname : \{0.1.0\} \rangle \end{array} \right\}$$

By forming the cross-product of leaf identifiers in the mapping C , the `matchSchema` algorithm computes the leaf candidates c . In our example, it is exactly one candidate c_1 :

$$c_1 = \langle city : 0.0, firstname : \{0.1.0\} \rangle$$

The algorithm uses the candidate c_1 to merge the root to leaf paths into a schema match candidate M_{c_1} :

$$M_{c_1} = \left\langle \begin{array}{l} \{0, 0.0, 0.1, 0.1.0\} \\ \{\langle 0, 0.0 \rangle, \langle 0, 0.1 \rangle, \langle 0.1.0 \rangle\} \end{array} \right\rangle$$

The candidate M_{c_1} has four nodes (upper line) and three edges (lower line). After successful validation, the algorithm transfers the constraints to obtain the

schema match M_1 as shown in Figure 4.2c. It adds M_1 to the set of schema matches \mathcal{M} , before returning \mathcal{M} , since there are no further candidates to be validated.

Once the algorithm has computed all schema matches, it starts the data matching phase.

4.3.2 Data matching

As shown in Algorithm 1, the algorithm calls the *matchData* function in Algorithm 3 after computing the schema matches \mathcal{M} to compute the tree-pattern matches on the data. It realizes the data-matching defined in Definition 4.8.

The *matchData* function has three different implementations to address the needs of the Pebble and Breadcrumb algorithms. They only differ in the return types:

- (i) The first type adds a boolean attribute to the input relation that indicates whether a top-level tuple in the input relation matches the tree-pattern. Breadcrumb leverages this *matchData* function to identify data that potentially contribute to the missing data in the result.
- (ii) The second type returns a nested relation of the type defined by the output nodes in the tree-pattern. It exclusively returns the data that matches the nodes in the tree-pattern with the output flag *on* set to true. The stand-alone tree-pattern matching algorithm leverages this function type to return the requested, matching data.
- (iii) The third type returns a relation that holds the exact paths the tree-pattern has matched on. Pebble leverages this type to compute the structural provenance for the data that match the tree-pattern.

Since the algorithm to compute the matches is conceptually the same for all three types, we exclusively describe the first variant here. In the implementation Chapter 7, we describe the implementation details for all three types.

Algorithm 3: *matchData*(\mathcal{M}, R)

Input: SchemaMatches \mathcal{M} , relation R of type \mathcal{R}

Output: Relation R' of type \mathcal{R}'

```
1  $R' \leftarrow \emptyset$ 
2 for  $t \in R$  do
3    $match \leftarrow \text{false}$ 
4   for  $M \in \mathcal{M}$  do
5      $match \leftarrow match \vee \text{checkTuple}(t, M.r, M.r)$ 
6    $R' \leftarrow R' \cup \{t \circ \langle match : match \rangle\}$ 
7 return  $R'$ 
```

The *matchData* function checks the value and cardinality constraints of each schema match $M \in \mathcal{M}$ against every tuple $t \in R$ to mark them as matching or non-matching. For that purpose, it associates an additional boolean attribute *match* to each t . It holds the true value, if t matches any $M \in \mathcal{M}$ and the false value otherwise.

After initializing the output collection R' to an empty relation \emptyset , Algorithm 3 iterates over each $t \in R$ (ll. 2-6) to check whether it matches. The variable *match* holds the boolean value indicating a match. The algorithm updates the variable while iterating over the schema matches $M \in \mathcal{M}$ (ll. 4-5) in a nested loop. It becomes true if any call to the *checkTuple* function returns true. This function recursively checks the value and cardinality constraints of every M against each tuple t . After checking all M s for a t , it concatenates t with the new attribute *match* and which holds the value of the variable *match* and adds the concatenated tuple to R' .

When the algorithm calls the recursive *checkTuple* function in Algorithm 4, it provides the parameters t , M 's root node r as initial node n , and $p = r$. In general, the function verifies that tuple t satisfies all the constraints of node n . The method keeps track of its nesting through path p . If the schema match node n , holds a value constraint, the function assesses the constraint on the value at $t.p$ (l. 3). If the value does not satisfy constraint vc_n of node n the function does not check any children of node n any more since t cannot

Algorithm 4: *checkTuple*(t, n, p)

Input: tuple t , node $n \in N_M$, path p

Output: boolean value indicating if t matches constraints of subtree of M rooted at n (true) or not (false)

```
1 isValid  $\leftarrow$  true
2 if  $vc_n \neq \perp$  then
3    $\lfloor$  isValid  $\leftarrow$  eval( $t.p, vc_n$ )
4 if isValid then
5   for  $c \in \text{childrenOf}(n)$  do
6     if  $c$  matches a collection type then
7        $\lfloor$  isValid  $\leftarrow$  isValid  $\wedge$  checkCol( $t, c, p.name_c$ )
8     else
9        $\lfloor$  isValid  $\leftarrow$  isValid  $\wedge$  checkTuple( $t, c, p.name_c$ )
10 return isValid;
```

match M anymore. If the value satisfies the constraint or if n has no value constraint $vc_n = \perp$, the function recursively checks n 's subtree in depth-first order. If n matches an attribute of primitive type, the recursion terminates because n does not have children in this case. Otherwise, the function iterates over n 's children c . During the iteration, the function distinguishes two cases for each child: (i) If the node c refers to an attribute of tuple or primitive type, the method calls itself to match the child node c reachable via the path $p.name_c$, where $name_c$ represents the attribute name. (ii) If the child node c refers to an attribute of the nested relation type, the algorithm further checks potential cardinality constraints on c . Then, it calls the *checkRel* method in Algorithm 5.

To verify cardinality constraints, the algorithm applies the *checkRel* function in Algorithm 5, which has the same parameters as the *checkTuple* function. The *checkRel* function first retrieves the nested relation REL from path p evaluated on t . Then, it iterates over the elements in REL , counting the nested tuples that match the subtree of M rooted in n . The function has to validate all constraints of the whole subtree. Thus, it calls *checkTuple* in

Algorithm 5: $checkRel(t, n, p)$

Input: tuple t , node $n \in N_M$, path p **Output:** boolean value indicating if t matches constraints of subtree of M rooted at n (true) or not (false)

```
1  $REL \leftarrow p^t$ 
2  $cnt \leftarrow 0$ 
3 for  $el \in REL$  do
4   if  $checkTuple(t, n, p[pos(el)])$  then
5      $cnt \leftarrow cnt + 1$ 
6  $isValid \leftarrow true$ 
7 if  $cc_n \neq \perp$  then
8    $isValid \leftarrow false$ 
9   if  $min_{cc} \leq cnt \leq max_{cc}$  then
10     $isValid \leftarrow true$ 
11 return  $isValid$ 
```

line 4 with an updated path pointing to a tuple in the nested relation. Only if all constraints on the tuple hold, it increments the counter cnt . If n has an associated cardinality constraint $cc_n \neq \perp$, the function checks that cnt is within the range defined by the cardinality constraint (ll. 4-6). Eventually, it returns `true` if the check was successful and `false` otherwise.

After the *matchData* algorithm in Algorithm 3 has checked all tree-patterns $M \in \mathcal{M}$ on all tuples in the input relation $t \in R$, the algorithm terminates. It has identified and marked all matching tuples utilizing the *checkTuple* function in Algorithm 4 and the *checkRel* function in Algorithm 5.

The following example illustrates the *matchData* algorithm on our running example.

Example 6

Given the set of schema matches \mathcal{M} , which contains the match M_1 computed in Section 4.3.1 and the tuple t in the example output Table 4.1, *matchData* starts checking t at M 's root node *root* with the path *root*. It will call the

functions introduced in this section multiple times. Hence, we use a superscript index that indicates the call order of these functions. It starts with calling the $checkTuple^1$ function with the specified parameters. Since there is no value constraint associated with the root node, the algorithm checks the child city next. It updates the path to $root.city$ and calls the $checkTuple^2$ with the city node and the updated path.

The city node holds the value constraint $vc_{city} = \langle =, LA \rangle$. Thus, the $checkTuple^2$ function compares the constraint's LA to the $root.city$ attribute's value LA. Since they match, $checkTuple^2$ advances to check the city node's children. The city node does not have any children. Thus, it returns *true* to $checkTuple^1$.

$checkTuple^1$ calls $checkCol^3$ on node $nList$ with path $root.nList$. The $checkCol^3$ function now checks the three tuples in the nested relation. It calls $checkTuple^4$ with node $firstname$ and path $root.nList[711].firstname$ to find the value Sue in $root.nList[711].firstname$ match the $firstname$'s value constraint. Thus, $checkTuple^4$ returns *true* to $checkCol^3$, which increases counter cnt by 1. $checkCol^3$ repeats calling $checkTuple^5$ with path $root.nList[712].firstname$ and $checkTuple^6$ with path $root.nList[713].firstname$. Like $checkTuple^4$, $checkTuple^5$ returns *true* since the value at path $root.nList[712].firstname$ is also Sue. $checkTuple^6$ returns *false* because the $firstname$ Tom does not match Sue.

After $checkCol^3$ has checked all nested tuples in $root.nList$, cnt equals 2. $checkCol^3$ compares the cnt to the upper and lower bound of the cardinality constraint associated with the $nList$ node. Both bounds are 2. Hence, $checkCol^3$ returns *true* to $checkTuple^1$, which also returns *true* after obtaining *true* from $checkCol^3$ and $checkTuple^2$. Thus, $matchData$ associates *true* to the input tuple t before adding it to the output relation R' .

4.3.3 Discussion

We conclude this section on the tree-pattern matching algorithm with a discussion on two crucial aspects of the algorithm. First, we discuss the

runtime complexity of the algorithm's two phases. Second, we elaborate on the distributed execution of the algorithm to highlight our contributions.

Runtime complexity. We first discuss the runtime complexity of the schema matching algorithm. Afterward, we focus on the data matching algorithm. Given a tree-pattern T with $n = |N|$ nodes, the schema matching algorithm (Algorithm 2) holistically computes schema matches \mathcal{M} over the data's schema, which has $r = |\mathcal{R}|$ attributes. More precisely, the nodes in the tree are either leaves l or inner nodes i . Hence, the number of nodes in the tree-pattern is the sum of leaves and inner nodes $n = n_l + n_i$. The algorithm first identifies all potential root-to-leaf paths in (ll. 6-8). Since the algorithm can leverage the DeweyID to check whether a leaf is the child of the root node, the loop can conduct the prefix check in constant time. In the worst case, the leaf node has the same attribute label as all attribute names in \mathcal{R} . Furthermore, the leaves in the tree may all share the same name. In this case, the complexity of this loop is in $O(r * n_l)$ because the algorithm has to check all attributes for all leaf nodes. The loop yields all root-to-leaf path matches for each leaf node. In the second loop (ll. 9-12), the algorithm iterates over all possible combinations of the previously computed root-to-leaf path matches. Hence, the loop is exponential in the number of leaf nodes. The root-to-leaf path matches are bound by r for each leaf node. Therefore, the loop introduces a complexity of $O(r^{n_l})$. In the loop, the algorithm calls the merge function that verifies whether the combination is a schema match. For each combination, the merge function has to check multiple node merge options, when ancestor-descendant relationships appear in the tree-pattern. In this case, the merge-function potentially has to match all inner nodes onto the merged tree. That has a worst-time complexity of $O(r^{n_i})$. Therefore, the second loop has a worst-time complexity of $O(r^{n_l} \cdot r^{n_i}) = O(r^{n_l+n_i}) = O(r^n)$. Combining the complexity of the first and second loop yields $O(r * n + r^n)$. Since the schema typically has more attributes than the tree-pattern, i.e., $r > n$, the runtime is dominated by the second loop and the worst-case complexity for the schema matching phase becomes $O(r^n)$. The average runtime complexity is much lower, though, because the worst-case complexity requires all nodes

in the tree-pattern and the schema to share the same labels, which is highly unlikely in practice.

Given the set of schema matches \mathcal{M} from the schema matching phase, the data matching phase computes the actual matches on the data. Let us assume the data holds d top-level tuples in the data and the number of schema matches is $m = |\mathcal{M}|$. The data matching phase accesses every tuple exactly once for each schema match. Hence, the data matching phase has a theoretical complexity of $O(m \cdot d)$. In practice, the number of schema matches is orders of magnitudes smaller than the number of top-level tuples in the data. Therefore, the complexity is close to $O(d)$.

In comparison, other holistic state-of-the-art algorithms like TJFast [LLCC05] or TwigStack [BKS02] match directly on the data. They enumerate all possible combinations of attribute instances for each node in the tree-pattern. Given d top-level tuples, which have up to r attributes each, they have a worst-case runtime complexity of $O((d \cdot r)^n)$. This is much worse than our runtime complexity since d typically is orders of magnitudes larger than r .

Scalability. As stated in the introduction (Contribution (2)), we contribute a novel tree-pattern matching algorithm that scales with the compute resources in a distributed big data analytics systems. Recall from the related work in Section 2.1 that existing algorithms do not simply extend to distributed systems because they rely on global state or produce large intermediate results. To avoid these limiting factors, our novel tree-pattern matching algorithm exploits the fact that the tuples in the nested relations of the analytics systems share the same schema. Hence, the algorithm splits the matching into two distinct phases. The schema matching phase runs single-threaded on a single compute node in the cluster. Distributing this phase will not improve the overall runtime since it computes schema matches exclusively on the schema, and the schema typically is orders of magnitude smaller than the data. Hence, we distribute the execution of the data matching phase.

Big data analytics systems typically partition the nested input relations into disjunct chunks of top-level tuples. Given the schema matches, the data matching function iterates over these top-level tuples to find the tree-

pattern matches. It checks each top-level tuple independently of all other top-level tuples in the nested relation. Thus, we can execute the function on the individual partitions as well as on the entire nested relation without influencing the result. That makes the algorithm scale to large dataset sizes on increasing distributed compute resources. We will show these properties in the evaluation chapter in Section 8.2. Here, we conclude the chapter with a summary.

4.4 Summary

In this chapter, we addressed our Contribution (2). For that purpose, we have introduced the tree-pattern syntax. We have further defined when a tree-pattern matches the data that complies with our data model. Based on these definitions, we have introduced a novel, scalable tree-pattern matching algorithm for big data analytics systems. This algorithm already improves the query capabilities of big data analytics systems, for it allows to query arbitrary combinations of nested data values. However, it shows its full potential only when leveraged in Pebble and Breadcrumb. They use the tree-pattern matching algorithm to compute explanations for existing data and missing data, respectively. In the next chapter, we will introduce the explanations for existing data and Pebble in detail.

CHAPTER



EXPLANATIONS FOR EXISTING DATA

Big data analytics systems lack sophisticated means to debug analytical queries, especially when the queries process nested data. That is why we propose structural provenance as new means to obtain explanations for existing data in the query result. These explanations point at data in the query input that are involved in producing the data in the result. The explanations are more comprehensive than those of comparable state-of-the-art solutions because they describe the access to and the manipulation of individual nested attributes. To provide these comprehensive explanations, the structural provenance captures structural manipulation of individual data values in addition to mere data dependencies. Since our goal is to efficiently compute the explanations obtained from structural provenance in a big data analytics system, we further contribute the Pebble algorithm that efficiently captures the structural provenance for nested input relations. This algorithm scales to large datasets in distributed clusters.

This chapter addresses Contribution (3) and describes the Pebble algorithm module in our architecture in Section 3.3. It is further based on two published papers [DH19; DH20b] and has the following structure. In Section 5.1, we formally introduce the structural provenance and its capture semantics. For that purpose, we extend the operator execution rules from our execution model in Table 3.1 to additionally capture data dependencies and the access to and manipulation of the data. The extended rules capture highly redundant information. That prevents capturing full-blown structural provenance at a large scale. Therefore, we introduce lightweight structural provenance in Section 5.2. It aggregates redundant access and manipulation information in the structural provenance so that it scales to large dataset sizes. Our novel Pebble algorithm captures this lightweight provenance while the big data analytics system executes the query. In Section 5.3, we describe how Pebble constructs the explanations from the captured provenance once an explanation is requested and discuss Pebble’s runtime complexity and Pebble’s scalability features in distributed big data analytics systems.

5.1 Structural provenance capture semantics

We introduce structural provenance to compute explanations for existing data. To provide more comprehensive explanations than other existing provenance, the structural provenance distinguishes between access and manipulation of data. Operators manipulate data when they change their attribute structure, attribute name, or data value. For instance, the tuple flatten operator unnests attributes. Hence, it manipulates the structure of these attributes. The rename operator replaces attribute names with new attribute names. The aggregation operator computes new data values from existing data values. All three example operators manipulate the data and the structural provenance captures these manipulations. Furthermore, operators access data during processing, but do not change the data structure, name, or value. For example, the selection operator accesses the data in the filter

condition to judge whether a tuple passes the selection or not. It does not manipulate the data structure, attribute names, or data values.

The structural provenance annotates top-level tuples in the nested relations that the operators in an analytical query process. It consists of multiple components to faithfully capture all access to and manipulation to arbitrarily nested data. We will introduce these components one after the other using color coding to improve understandability. Afterward, we will define the provenance capture rules that describe the provenance capture semantics.

To capture the data access, the structural provenance records the access provenance for each top-level tuple in each operator's input relation. It records all paths accessed in each tuple.

Definition 5.1 (Access provenance)

Let R_i be the input relations of an operator O and $t \in R_i$ be a top-level input tuple. Then, t 's access provenance \mathcal{A}_t is the set of paths that the operator O accesses to obtain the corresponding data values in t . O needs the values to process t . The access provenance stores each accessed path in a tuple with a single attribute called p .

$$\mathcal{A}_t := \bigcup_{t.p \text{ accessed by } O} \{\{p : t.p\}\}$$

$$\text{type}(\mathcal{A}_t) := \{\{p : STR\}\}$$

The access provenance is part of the input provenance, which is a nested relation. Intuitively, this relation holds all top-level input tuples that contribute to an output tuple together with their associated access provenance, and their input relation.

Definition 5.2 (Input provenance)

Let R_i be the input relations of an operator O , t be a top-level input tuple $t \in R_i$, and t' be a top-level output tuple $t' \in \llbracket O \rrbracket$. Then, t' 's input provenance $\mathcal{I}_{t'}$ is a

nested relation with one 3-tuple for each input tuple t that contributes to t' :

$$\mathcal{I}_{t'} := \bigcup_{t \text{ contributes to } t'} \{ \langle it : id(t), iR : id(R_i), \mathcal{A} : \mathcal{A}_t \rangle \}$$

$$\text{type}(\mathcal{I}_{t'}) := \{ \langle it : INT, iR : INT, \mathcal{A} : \text{type}(\mathcal{A}_t) \rangle \}$$

The 3-tuple has the identifier of the input tuple $id(t)$, which is unique within t 's nested relation R_i . The tuple also references the nested relation R_i via its globally unique identifier $id(R_i)$ and stores the access provenance \mathcal{A}_t .

The input provenance is sufficient to track dependencies between top-level tuples in the input and the output of an operator. They do not record any manipulations of nested attributes. We track them in the manipulation provenance so that the structural provenance can yield explanations at the granularity of individual nested attributes. The manipulation provenance is a mapping between paths in top-level input tuples and paths in top-level output tuples. For each input tuple that helps to produce an output tuple, the mapping stores an entry for each attribute path in the input tuple that helps to produce an attribute path in the output tuple. More generally, if input data helps to produce output data, we say that the input data contributes to the output data.

Definition 5.3 (Manipulation provenance)

Let R_i be the input relations of an operator O , t be a top-level input tuple $t \in R_i$, and t' be a top-level output tuple $t' \in \llbracket O \rrbracket$. Then, t' 's manipulation provenance $\mathcal{M}_{t'}$ is a nested relation that holds one tuple for each input attribute path $t.p$ that contributes an output attribute path $t'.p'$:

$$\mathcal{M}_{t'} := \bigcup_{t.p \text{ contributes to } t'.p'} \{ \langle in : t.p, out : t'.p' \rangle \}$$

$$\text{type}(\mathcal{M}_{t'}) := \{ \langle in : STR, out : STR \rangle \}$$

The manipulation provenance and the input provenance together form the structural provenance captured during query execution. It is associated with each top-level output tuple.

Definition 5.4 (Structural provenance capture extension)

Let t' be a top-level output tuple $t' \in \llbracket O \rrbracket$ and $\mathcal{I}, \mathcal{M} \notin \mathbb{L}$ unique attribute labels. The structural provenance capture extension \mathcal{SP} and its type $\text{type}(\mathcal{SP})$ are:

$$\begin{aligned}\mathcal{SP}_{t'} &:= \langle \mathcal{I} : \mathcal{I}_{t'} \rangle \circ \langle \mathcal{M} : \mathcal{M}_{t'} \rangle \\ \text{type}(\mathcal{SP}_{t'}) &:= \langle \mathcal{I} : \text{type}(\mathcal{I}_{t'}), \mathcal{M} : \text{type}(\mathcal{M}_{t'}) \rangle\end{aligned}$$

The capture extension's type is independent of the output type $\text{type}(t')$. In fact, we have designed the type so generic that it applies to the output of all supported operators. The capture extension for each top-level output tuple is computed when an operator is executed.

Definition 5.5 (Structural provenance execution)

Let $t' \in \llbracket O \rrbracket$ of type $\tau_{t'}$ be a top-level tuple in the output of an executed operator and $\mathcal{SP}_{t'}$ its provenance capture extension. Then, the structural provenance execution $\llbracket O \rrbracket^{\mathcal{SP}}$ is:

$$\begin{aligned}\llbracket O \rrbracket^{\mathcal{SP}} &:= \{\{t' \circ \mathcal{SP}_{t'} \mid t' \in \llbracket O \rrbracket\}\} \\ \text{type}(\llbracket O \rrbracket^{\mathcal{SP}}) &:= \{\{\tau_{t'} \circ \text{type}(\mathcal{SP}_{t'})\}\}\end{aligned}$$

In the following, we extend the operator execution rules in our execution model (cf. Table 3.1) with the capabilities to capture the structural provenance \mathcal{SP} . For conciseness, we do not repeat the preconditions of the operator semantics, even though they also apply to the following rules. We denote top-level tuples in the operator's input relations with t and the corresponding top-level tuples in the output relation with t' . The extended rules typically have four lines. They have the output tuple t' in the first line, the input provenance \mathcal{I} in the second line, and the manipulation provenance \mathcal{M} in the third line. The access provenance \mathcal{A} , which is part of \mathcal{I} , and the manipulation provenance \mathcal{M} are highlighted for convenience. The fourth line describes how the output tuple t' is generated from the input tuples. We do not show

the output type for conciseness, since the provenance type is defined by the generic definition above. We start with the table access operator.

Table access. The table access operator reads the only input relation R without accessing particular attributes or manipulating the data. Thus, the access provenance \mathcal{A} and the manipulation provenance \mathcal{M} are both empty, denoted as \emptyset .

$$\begin{aligned} \llbracket R \rrbracket^{SP} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \{ \{ \langle it : id(t), iR : id(R), \mathcal{A} : \emptyset \rangle \} \} \rangle \\ & \circ \langle \mathcal{M} : \emptyset \rangle \\ & | t' = t \wedge t \in R \} \} \end{aligned}$$

Accessing the table preserves the multiplicity of all tuples. Thus, $t'^m \in \llbracket R \rrbracket^{SP}$ holds, if $t^m \in R$.

Projection. Recall that the projection operator projects each input tuple t to a new output tuple t' that has precisely the attributes L . Therefore, it accesses all attributes with a label $l \in L$ to add the attribute to the output tuple t' . Thus, \mathcal{A} records all paths $t.l$ and \mathcal{M} the mapping between the input path $t.l$ and the output path $t'.l$.

$$\begin{aligned} \llbracket \pi_L(R) \rrbracket^{SP} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \{ \{ \langle it : id(t), iR : id(R), \mathcal{A} : \bigcup_{l \in L} \{ \{ t.l \} \} \} \} \} \rangle \\ & \circ \langle \mathcal{M} : \bigcup_{l \in L} \{ \{ \langle in : t.l, out : t'.l \rangle \} \} \rangle \\ & | t' = t.L \wedge t \in R \} \} \end{aligned}$$

The multiplicity of t' is t'^m where $m = \sum_{t:t.L=t'} \text{MULT}(R, t)$.

Renaming. The renaming operator renames attributes of top-level input tuples t . Each input tuple yields exactly one output tuple t' with the renamed attributes. The input provenance \mathcal{I} is similar to the the projection's input provenance. Instead of referencing the attribute labels that the projection

obtains as parameters, the renaming references all attribute labels that appear in t . The manipulation provenance \mathcal{M} also resembles the projection's manipulation provenance. Instead of recording the same paths for input and output, the \mathcal{M} records the old path with attribute label A_i for the input tuple and the path with the new attribute name B_i for the output tuple.

$$\begin{aligned} \llbracket \rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R) \rrbracket^{\mathcal{SP}} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \{ \{ \langle it : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup_{i \in \{1, \dots, n\}} \{ \{ t.A_i \} \} \} \} \rangle \rangle \\ & \circ \langle \mathcal{M} : \bigcup_{i \in \{1, \dots, n\}} \{ \{ \langle in : t.A_i, out : t'.B_i \rangle \} \} \rangle \\ & | t \in R \wedge t' = \langle B_1 : t'.A_1, \dots, B_n : t'.A_n \rangle \} \} \end{aligned}$$

If the multiplicity of $t \in R$ is t^l , then t' occurs l times in the output: t'^l .

Selection. The selection operator removes all tuples from the input relation R that do not satisfy the selection condition θ . It does not manipulate the structure or values of the tuples, thus \mathcal{M} is empty. However, the selection operator accesses all paths occurring in the filter condition θ for each input tuple t . The access provenance \mathcal{A} records these accesses.

$$\begin{aligned} \llbracket \sigma_{\theta}(R) \rrbracket^{\mathcal{SP}} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \{ \{ \langle it : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup t.p \in \theta \rangle \} \} \rangle \rangle \\ & \circ \langle \mathcal{M} : \emptyset \rangle \\ & | t' = t \wedge t \in R \wedge t' \models \theta \} \} \end{aligned}$$

The multiplicity of all tuples t' in the output t'^k is equal to the multiplicity in the input relation $t^k \in R$ if t satisfies the filter condition and 0, otherwise.

Join. The join operator has four variants. We extend the inner join semantics with the structural provenance capture. Afterward, we informally describe the structural provenance capture for the three outer join variants.

- **Inner join.** The inner join operator concatenates the tuples from two input relations $t_1 \in R$ and $t_2 \in S$ to an output tuple t' if t' satisfies the join condition, denoted as $t' \models \theta$. The input provenance \mathcal{I} holds two records per output tuple. These records point to the two contributing tuples in each of the input relations. Further, their access provenance \mathcal{A} records all paths accessed in t_1 and t_2 to evaluate the condition θ . It resembles the access provenance in the selection operator. The manipulation provenance \mathcal{M} records mappings between all attributes in t_1 and t_2 to their corresponding attributes in t' . For that purpose, it leverages the label function $\text{LBL}(R)$ that returns the attribute labels in $\text{LBL}(R)$.

$$\begin{aligned} \llbracket R \bowtie_{\theta} S \rrbracket^{\mathcal{SP}} := \{ \{ t' \} \\ \circ \langle \mathcal{I} : \{ \{ \langle it:\text{id}(t_1), iR:\text{id}(R), \mathcal{A} : \bigcup \{ \{ t_1.p \in \theta \} \} \rangle, \langle it:\text{id}(t_2), iS:\text{id}(S), \mathcal{A} : \bigcup \{ \{ t_2.p \in \theta \} \} \rangle \} \} \rangle \\ \circ \langle \mathcal{M} : \bigcup_{l \in \text{LBL}(R)} \{ \langle in:t_1.l, out:t'.l \rangle \} \cup \bigcup_{l \in \text{LBL}(S)} \{ \langle in:t_2.l, out:t'.l \rangle \} \} \\ | t' = t_1 \circ t_2 \wedge t_1 \in R \wedge t_2 \in S \wedge t_1 \circ t_2 \models \theta \} \end{aligned}$$

The multiplicity of t' is $t'^{k \cdot m}$, where $t_1^k \in R \wedge t_2^m \in S$.

- **Outer joins.** The inner join only adds concatenated tuples to the result when two tuples in the input become join partners. The **left outer join**, the **right outer join**, and the **full outer join** also yield tuples in the result that originate from one input tuple only. For these output tuples, the input provenance holds just a single record that refers to the only contributing input tuple. Further, the manipulation provenance captures only those path mappings that originate in the contributing input tuple.

Flatten. The flatten operator flattens a nested tuple or a nested relation. We extend the rules for the tuple flatten operator and the inner flatten

operator with structural provenance capture. Then, we informally describe how to modify the rule for outer flatten based on the extended inner flatten operator.

- **Tuple flatten.** The tuple flatten operator unnests an attribute A of type \mathcal{T} . If A is of tuple type: $\tau = \langle B_1 : \tau'_1, \dots, B_m : \tau'_m \rangle$, then the tuple flatten $F_A^T(R)$ operator semantics with structural provenance collection are:

$$\begin{aligned} \llbracket F_A^T(R) \rrbracket^{\mathcal{SP}} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \{ \{ \langle it : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup_{l \in \text{LBL}(t.A)} \{ \{ t.A.l \} \} \} \} \rangle \\ & \circ \langle \mathcal{M} : \bigcup_{l \in \text{LBL}(t.A)} \{ \{ \langle in : t.A.l, out : t'.l \} \} \} \\ & | t' = t \circ t.A \wedge t \in R \} \} \end{aligned}$$

The output tuple t'^k has the same multiplicity as the input tuple $t^k \in R$, from which t' is generated. The input provenance \mathcal{I} has exactly one entry with t for each t' . Further, it keeps track of all paths accessed in the attribute $t.A$ in the access provenance since the flatten operator accesses them to flatten out the nested tuple in $t.A$. The manipulation provenance \mathcal{M} holds mappings between the paths to the attributes in the nested tuple $t.A.l$ and the attributes in $t'.l$ to account for the manipulated structure.

- **Inner flatten.** The inner flatten operator concatenates each top-level input tuple $t \in R$ with the tuple u in the nested relation attribute $t.A$. If the attribute $t.A$ has the null value \perp or an empty relation \emptyset , t does not contribute to the output.

$$\begin{aligned}
\llbracket F_A^I(R) \rrbracket^{SP} := & \{ \{ t' \\
& \circ \langle \mathcal{I} : \{ \{ \langle ir : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup_{l \in \text{LBL}(t.A)} \{ \{ t.A[\text{id}(u)].l \} \} \} \} \rangle \\
& \circ \langle \mathcal{M} : \bigcup_{l \in \text{LBL}(t.A)} \{ \{ \langle in : t.A[\text{id}(u)].l, out : t'.l \} \} \} \rangle \\
& | t' = t \circ u \wedge t \in R \wedge u \in t.A \}
\end{aligned}$$

The multiplicity of t' is $t'^{k.l}$, where $t^k \in R$ and $u^l \in t.A$. Each output tuple t' originates in one input tuple t . Thus, the input provenance \mathcal{I} has one record referencing t . Its access Provenance \mathcal{A} records all paths to the attributes in u , which the operator accesses to flatten $t.A$. The manipulation provenance \mathcal{M} holds mappings between the attributes in u and the newly added attributes in t' .

- **Outer flatten.** The outer flatten operator resembles the inner flatten operator. However, it preserves tuples t whose attribute $t.A$ has the null value \perp or an empty relation \emptyset . The access and manipulation provenance are empty for these tuples because the operator cannot access attributes in non-existing tuples u .

Nesting. The nesting operator nests attributes into nested tuples or relations. We extend both versions of the nesting with structural provenance.

- **Tuple nesting.** The tuple nesting operator $\mathcal{N}_{N \rightarrow C}^T$ constructs a tuple from the nesting attributes N and nests this tuple into a new tuple-typed attribute C . It concatenates the new attribute C with all attributes $G = \text{LBL}(t) - N$ for the result.

$$\begin{aligned}
\llbracket \mathcal{N}_{A \rightarrow C}^T(R) \rrbracket^{\mathcal{SP}} := & \{ \{ t' \\
& \circ \langle \mathcal{I} : \{ \{ \langle it : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup_{l \in N} \{ \{ t.l \} \} \} \} \rangle \\
& \circ \langle \mathcal{M} : \bigcup_{l \in N} \{ \{ \langle in : t.l, out : t'.C.l \} \} \} \rangle \\
& | t' = t.G \circ \langle C : t.G \rangle \wedge t \in R \wedge G = \text{LBL}(t) - N \} \}
\end{aligned}$$

The output tuple t' has the multiplicity of t'^k , if the input tuple t has the multiplicity $t^k \in R$. The input provenance \mathcal{I} holds a reference to the input tuple t that t' is derived from. Additionally, it holds the paths to all attributes in $t.N$ in its access provenance \mathcal{A} because they are accessed to form the new nested tuple. The manipulation provenance \mathcal{M} records the mappings between the attributes in $t.N$ and the newly nested attributes in $t'.C$.

- **Relation nesting.** The relation nesting $\mathcal{N}_{N \rightarrow C}^R$ operator groups the input tuples based on the grouping attributes $G = \text{LBL}(t) - N$. For each group, it yields a tuple with the attributes mentioned in G plus a new attribute C of relation type that contains all tuples in the group. Before the operator nests these tuples, it applies a projection on the tuples on the nesting attributes N .

$$\begin{aligned}
\llbracket \mathcal{N}_{N \rightarrow C}^R(R) \rrbracket^{SP} &:= \{t'\} \\
&\circ \langle \mathcal{I}: \bigcup_{t'' \in ig(R,G,t)} \langle it: id(t''), iR: id(R), \mathcal{A}: \bigcup_{l \in LBL(t'')} \{\{t''.l\}\} \rangle \rangle \\
&\circ \left\langle \mathcal{M}: \bigcup_{t'' \in ig(R,G,t)} \left\{ \bigcup_{l \in N} \{\{in: t''.l, out: t'.C[id(t'')].l\}\} \right\} \right\rangle \\
&|t' = (t.G \circ ns(R,G,N,C,t))^1 \wedge t \in gr(R,G) \wedge G = LBL(t) - N \} \\
gr(R,G) &:= \{t.G | t^n \in R\} \\
ig(R,G,t) &:= \llbracket (\sigma_{t'.G=t.G}(\{t' | t'^n \in R\})) \rrbracket \\
ns(R,G,N,C,t) &:= \langle C: \llbracket \pi_N(ig(R,G,t)) \rrbracket \rangle
\end{aligned}$$

Each t' occurs exactly once in the output, as indicated by the superscript 1 in $t' = (t.G \circ ns(R, G, N, C, t))^1$. The definition of the relation nesting operator makes use of the three helper functions $gr(R, G)$, $ig(R, G, t)$, and $ns(R, G, N, C, t)$. Recall that $gr(R, G)$ groups the tuples by the attributes G , and that $ns(R, G, N, C, t)$ creates the attribute C with the new nested relation. For the provenance collection, the operator semantics make use of the additional $ig(R, G, t)$ function, which yields all tuples in the group of t .

The relation nesting operator has one record for each input tuple t'' in the input provenance \mathcal{I} that is in the group of tuples from which the operator computes t' . For each of the input tuples t'' , the accesses provenance \mathcal{A} marks all attributes of t'' as accessed because the operator accesses the attributes in G to assess the group of t'' and the attributes in N to form the tuple to be added to the new nested relation in $t'.C$. Further, $G \cup N = LBL(t'')$ holds. The manipulation provenance \mathcal{M} holds mappings for each input tuple t'' 's nesting attributes in $t''.N$, which map to the corresponding attribute in the nested relation $t'.C$. By definition, the identifier of the input tuple t'' also is the identifier of the newly created tuples in $t.C$.

Aggregation. The aggregation operator $\gamma_{f(A) \rightarrow B}$ aggregates the values in a nested relation $t.A$ into a new attribute B of primitive type for each input tuple t .

$$\begin{aligned} \llbracket \gamma_{f(A) \rightarrow B}(R) \rrbracket^{SP} := & \{ \{ t' \\ & \circ \langle \mathcal{I} : \langle it : \text{id}(t), iR : \text{id}(R), \mathcal{A} : \bigcup_{u \in t.A} \{ \{ t.A[\text{id}(u)] \} \} \rangle \rangle \\ & \circ \langle \mathcal{M} : \bigcup_{u \in U \subseteq t.A} \{ \langle in : t.A[\text{id}(u)], out : t'.B \rangle \} \} \} \\ & | t' = t \circ \langle B : f(t.A) \rangle \wedge t \in R \} \end{aligned}$$

The multiplicity of the output tuple t'^k is equal to the multiplicity of the input tuple $t^k \in R$. Note that U is the subset of nested tuples in $t.A$, that $f(t.A)$ computes the aggregated value from. By default, $U = t.A$, which yields the correct provenance for *sum*, *avg*, or *count* functions. However, for other functions, such as *min* or *max*, U may be the true subset of tuples that hold the minimum or maximum values, respectively.

The input provenance \mathcal{I} holds a reference to the single input tuple $t \in R$. It adds the paths to the tuples u nested in the relation $t.A$ to its access provenance \mathcal{A} . The manipulation provenance \mathcal{M} keeps record of those tuples that actually contribute to the value in $t'.B$. For that purpose it holds mappings for all nested tuples in U to $t'.B$.

Union. The union operator merges two input relations R and S that share the same schema, i.e., $\text{type}(R) = \text{type}(S)$, into a single output relation. It does not access any attribute values. Neither does it manipulate any data structure of the tuples in R and S . Thus, the union operator with structural provenance extension can reuse the extended table access operator to get the provenance extensions \mathcal{I} and \mathcal{M} in a first step. Once R and S are extended with their structural provenance, the regular union operator can merge the extended R and S into a single output relation.

$$\llbracket R \cup S \rrbracket^{SP} := \llbracket R \rrbracket^{SP} \cup \llbracket S \rrbracket^{SP}$$

The provenance extension has no impact on the multiplicities in the output relation. The multiplicities defined in $\llbracket R \cup S \rrbracket$ also apply here.

We illustrate the extensions introduced by the structural provenance on our running example from Section 1.2.

Example 7

We execute the flatten operator in the example pipeline $F_{address1}^I$ on the shortened example input relation R_{short} shown in Table 5.1. Unlike the full input in Table 1.1, the shortened example input only holds the two “Sue” tuples from Table 1.1. The output of the executed flatten operator $\llbracket F_{address1}^I(R_{short}) \rrbracket^{SP}$ is shown in Table 5.2.

The output consists of four tuples. As the input provenance \mathcal{I} indicates, the flatten operator generates the first two tuples with identifier 113 and 114 from the input tuple with identifier 2. Analogously, it computes the last two tuples with identifier 115 and 116 from the last input tuple with identifier 3. The input provenance \mathcal{I} further captures the nested tuple that the flatten operator concatenates with the top-level input tuple. The paths in \mathcal{I} ’s access provenance show that the flatten operator accesses the *city* and *year* attribute of tuple 31 in the nested relation *address1*.

The manipulation provenance \mathcal{M} further reveals that the *city* and *year* attribute in the output tuple 113 originate from tuple 31 in the nested relation *address1*.

In summary, the input and manipulation provenance in Table 5.2 show that each tuple in the output is computed from a different pair of a top-level tuple and a tuple nested in its *address1* attribute. It further captures the origins of the unnested attributes *city* and *year*.

The above example shows that the structural provenance stores dependencies for individual input and output attributes in nested relations. The example also illustrates that the structural provenance stores a lot of redundant information when faithfully implemented as described in this section. For instance, the captured paths in the access provenance point at actual values in the input data. They only distinguish in the identifiers of the nested tuples. The following section generalizes this observation to derive

id	firstname	lastname	address1			address2		
2	Sue	Miller	city	year		city	year	
			31	LA	2019	41	NY	2019
			32	NY	2018	42	LA	2018
3	Sue	Walker	city	year		city	year	
			51	SF	2018	61	LV	2017
			52	LA	2019	62	NY	2019

Table 5.1: The example input relation R_{short} , which is the example relation from Table 1.1 reduced to the two “Sue” tuples; italic numbers indicate tuple identifiers

a lightweight provenance capture extension. This extension utilizes the prospective provenance introduced in Section 2.2 to minimize redundant information.

firstname	...	address1	...	city	year	\mathcal{I}	\mathcal{M}																					
113	Sue	...	<table border="1"> <thead> <tr><th>city</th><th>year</th></tr> </thead> <tbody> <tr><td>31</td><td>LA 2019</td></tr> <tr><td>32</td><td>NY 2018</td></tr> </tbody> </table>	city	year	31	LA 2019	32	NY 2018	...	LA 2019	<table border="1"> <thead> <tr><th>it</th><th>iR</th><th>\mathcal{A}</th></tr> </thead> <tbody> <tr><td>2</td><td><i>R_{short}</i></td><td> <table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[31].city</td></tr> <tr><td>address1[31].year</td></tr> </tbody> </table> </td></tr> </tbody> </table>	it	iR	\mathcal{A}	2	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[31].city</td></tr> <tr><td>address1[31].year</td></tr> </tbody> </table>	p	address1[31].city	address1[31].year	<table border="1"> <thead> <tr><th>in</th><th>out</th></tr> </thead> <tbody> <tr><td>address1[31].city</td><td>city</td></tr> <tr><td>address1[31].year</td><td>year</td></tr> </tbody> </table>	in	out	address1[31].city	city	address1[31].year	year
city	year																											
31	LA 2019																											
32	NY 2018																											
it	iR	\mathcal{A}																										
2	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[31].city</td></tr> <tr><td>address1[31].year</td></tr> </tbody> </table>	p	address1[31].city	address1[31].year																							
p																												
address1[31].city																												
address1[31].year																												
in	out																											
address1[31].city	city																											
address1[31].year	year																											
114	Sue	...	<table border="1"> <thead> <tr><th>city</th><th>year</th></tr> </thead> <tbody> <tr><td>31</td><td>LA 2019</td></tr> <tr><td>32</td><td>NY 2018</td></tr> </tbody> </table>	city	year	31	LA 2019	32	NY 2018	...	NY 2018	<table border="1"> <thead> <tr><th>it</th><th>iR</th><th>\mathcal{A}</th></tr> </thead> <tbody> <tr><td>2</td><td><i>R_{short}</i></td><td> <table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[32].city</td></tr> <tr><td>address1[32].year</td></tr> </tbody> </table> </td></tr> </tbody> </table>	it	iR	\mathcal{A}	2	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[32].city</td></tr> <tr><td>address1[32].year</td></tr> </tbody> </table>	p	address1[32].city	address1[32].year	<table border="1"> <thead> <tr><th>in</th><th>out</th></tr> </thead> <tbody> <tr><td>address1[32].city</td><td>city</td></tr> <tr><td>address1[32].year</td><td>year</td></tr> </tbody> </table>	in	out	address1[32].city	city	address1[32].year	year
city	year																											
31	LA 2019																											
32	NY 2018																											
it	iR	\mathcal{A}																										
2	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[32].city</td></tr> <tr><td>address1[32].year</td></tr> </tbody> </table>	p	address1[32].city	address1[32].year																							
p																												
address1[32].city																												
address1[32].year																												
in	out																											
address1[32].city	city																											
address1[32].year	year																											
115	Sue	...	<table border="1"> <thead> <tr><th>city</th><th>year</th></tr> </thead> <tbody> <tr><td>51</td><td>SF 2018</td></tr> <tr><td>52</td><td>LA 2019</td></tr> </tbody> </table>	city	year	51	SF 2018	52	LA 2019	...	SF 2018	<table border="1"> <thead> <tr><th>it</th><th>iR</th><th>\mathcal{A}</th></tr> </thead> <tbody> <tr><td>3</td><td><i>R_{short}</i></td><td> <table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[51].city</td></tr> <tr><td>address1[51].year</td></tr> </tbody> </table> </td></tr> </tbody> </table>	it	iR	\mathcal{A}	3	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[51].city</td></tr> <tr><td>address1[51].year</td></tr> </tbody> </table>	p	address1[51].city	address1[51].year	<table border="1"> <thead> <tr><th>in</th><th>out</th></tr> </thead> <tbody> <tr><td>address1[51].city</td><td>city</td></tr> <tr><td>address1[51].year</td><td>year</td></tr> </tbody> </table>	in	out	address1[51].city	city	address1[51].year	year
city	year																											
51	SF 2018																											
52	LA 2019																											
it	iR	\mathcal{A}																										
3	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[51].city</td></tr> <tr><td>address1[51].year</td></tr> </tbody> </table>	p	address1[51].city	address1[51].year																							
p																												
address1[51].city																												
address1[51].year																												
in	out																											
address1[51].city	city																											
address1[51].year	year																											
116	Sue	...	<table border="1"> <thead> <tr><th>city</th><th>year</th></tr> </thead> <tbody> <tr><td>51</td><td>SF 2018</td></tr> <tr><td>52</td><td>LA 2019</td></tr> </tbody> </table>	city	year	51	SF 2018	52	LA 2019	...	LA 2019	<table border="1"> <thead> <tr><th>it</th><th>iR</th><th>\mathcal{A}</th></tr> </thead> <tbody> <tr><td>3</td><td><i>R_{short}</i></td><td> <table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[52].city</td></tr> <tr><td>address1[52].year</td></tr> </tbody> </table> </td></tr> </tbody> </table>	it	iR	\mathcal{A}	3	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[52].city</td></tr> <tr><td>address1[52].year</td></tr> </tbody> </table>	p	address1[52].city	address1[52].year	<table border="1"> <thead> <tr><th>in</th><th>out</th></tr> </thead> <tbody> <tr><td>address1[52].city</td><td>city</td></tr> <tr><td>address1[52].year</td><td>year</td></tr> </tbody> </table>	in	out	address1[52].city	city	address1[52].year	year
city	year																											
51	SF 2018																											
52	LA 2019																											
it	iR	\mathcal{A}																										
3	<i>R_{short}</i>	<table border="1"> <thead> <tr><th>p</th></tr> </thead> <tbody> <tr><td>address1[52].city</td></tr> <tr><td>address1[52].year</td></tr> </tbody> </table>	p	address1[52].city	address1[52].year																							
p																												
address1[52].city																												
address1[52].year																												
in	out																											
address1[52].city	city																											
address1[52].year	year																											

Table 5.2: Output of the flatten operator in the example pipeline Figure 1.1, when applied to R_{short} : $\llbracket F_{address1}^I(R_{short}) \rrbracket^{SP}$; italic numbers indicate tuple identifiers; the attributes lastname and address2 are hidden for conciseness

5.2 Lightweight provenance capture

The provenance capture extensions introduced in the previous section have the potential for optimization. Since our goal is to provide a scalable and efficient Pebble algorithm, we leverage prospective provenance to minimize redundantly stored information in the capture extensions. We identify two potential optimization hooks.

First, the extensions store the paths accessed and manipulated repeatedly for each tuple in the output. They only differ in the identifiers of the top-level tuples and of the tuples in nested relations. Thus, recording the accessed and manipulated paths at schema level using placeholders for tuple identifiers prevents recording similar paths repeatedly.

Second, the capture extensions have such a generic data structure that they accommodate all operators in the same way. Tailoring the extensions to individual operators further reduces redundancies in the capture structure. For instance, the selection operator generates each top-level output tuple from exactly one top-level input tuple. The join operator requires exactly two top-level input tuples from different input relations. Providing custom extensions for each operator type further improves scalability and efficiency, but requires tracking the operator type.

The lightweight provenance capture exploits the two observations to keep overhead at capture time low. It has two components, the lightweight schema extension \mathcal{LSP}_O^S , and the lightweight instance extension \mathcal{LSP}_O^I , which we define next.

Definition 5.6 (Lightweight schema extension)

Let O be an operator that computes the output relation $T = \llbracket O \rrbracket$ from the input relations $R_i \in \{R_1, \dots, R_n\}$. Further, let $\text{type}(O)$ yield the operator type. Then, the lightweight schema extension \mathcal{LSP}_O^S is the following 4-tuple:

$$\mathcal{LSP}_O^S = \langle oR : id(T), otype : \text{type}(O), \mathcal{I}^S : \bigcup_{R_i} \{ \langle iR : id(R_i), \mathcal{A}^S : \mathcal{A}_{R_i}^S \rangle, \mathcal{M}^S : \mathcal{M}_T^S \} \rangle$$

The oR attribute holds the identifier $id(T)$ of the output relation T . The $otype$ attribute holds the operator type of O . The attribute \mathcal{I}^S holds the input

provenance at schema level. It is a nested relation that contains one tuple for each input relation R_i . The tuple holds the identifier of the input relation $id(R_i)$ in the attribute iR . Additionally, it holds the attribute \mathcal{A}^S to record the schema-level access provenance $\mathcal{A}_{R_i}^S$ of the output relation T . \mathcal{A}^S resembles the access provenance \mathcal{A} of Definition 5.1. It records the accessed path on the schema of R_i and replaces tuple identifiers with placeholders. The manipulation provenance attribute \mathcal{M}^S resembles the manipulation provenance \mathcal{M} in Definition 5.3 like \mathcal{A}^S resembles \mathcal{A} . It holds paths on the schema level with placeholders for tuple identifiers.

The lightweight schema extension \mathcal{LSP}_O^S still has the same operator-independent structure for all operators. The following lightweight instance extension \mathcal{LSP}_O^I depends on the operator type.

Definition 5.7 (Lightweight instance extension)

Let operator O compute the output relation $T = \llbracket O \rrbracket$ from the input relations $R_i \in \{R_1, \dots, R_n\}$. Further, let $t' \in T$, $t, t_1 \in R_1$, $t_2 \in R_2$, and u be an element of nested relation in t according to the operator semantics defined in Section 5.1. Then, the operator-dependent lightweight instance extensions \mathcal{LSP}_O^I are defined in Table 5.3.

The table access, projection, renaming, selection, tuple flatten, and tuple nesting operator hold pairs of a single input tuple identifier $in : id(t)$ and a single output tuple identifier $out : id(t')$ because they all have exactly one input relation that computes each output tuple from a single input tuple. The join and the union operator have two input tuple identifiers lin and rin for each output tuple identifier to account for the two input relations. The relation flatten operator holds a $nest$ identifier in addition to the in and out identifiers to reference the nested tuple, from which the output tuple is generated. The relation nesting operator holds a nested relation $inRel$ that has the identifiers of all input tuples which form the output tuple. The aggregation operator holds a nested relation $nRel$ which holds references to tuples in the attribute, to which the aggregate function is applied. $nRel$ only

Operator	Lightweight instance extension
table access	$\mathcal{LSP}_R^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
projection	$\mathcal{LSP}_\pi^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
renaming	$\mathcal{LSP}_\rho^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
selection	$\mathcal{LSP}_\sigma^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
join	$\mathcal{LSP}_\bowtie^I = \{\{\langle lin : id(t_1), rin : id(t_2), out : id(t') \rangle\}\}$
tuple flatten	$\mathcal{LSP}_{FT}^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
relation flatten	$\mathcal{LSP}_{FIO}^I = \{\{\langle in : id(t), nest : id(u), out : id(t') \rangle\}\}$
tuple nesting	$\mathcal{LSP}_{NT}^I = \{\{\langle in : id(t), out : id(t') \rangle\}\}$
relation nesting	$\mathcal{LSP}_{NR}^I = \{\{\langle inRel : \{\{\langle in : id(t) \rangle\}\}, out : id(t') \rangle\}\}$
aggregation	$\mathcal{LSP}_\gamma^I = \{\{\langle in : id(t), nRel : \{\{\langle nest : id(u) \rangle\}\}, out : id(t') \rangle\}\}$
union	$\mathcal{LSP}_\cup^I = \{\{\langle lin : id(t_1), rin : id(t_2), out : id(t') \rangle\}\}$

Table 5.3: Operator-dependent lightweight instance extension \mathcal{LSP}^I

holds identifiers of those nested tuples that contribute to the aggregated output value.

The schema extension \mathcal{LSP}_O^S and the instance extension \mathcal{LSP}_O^I of an operator O form the lightweight structural provenance \mathcal{LSP}_O .

Definition 5.8 (Lightweight structural provenance)

Given an operator O , its lightweight structural provenance is:

$$\mathcal{LSP}_O := \langle \mathcal{LSP}^S : \mathcal{LSP}_O^S, \mathcal{LSP}^I : \mathcal{LSP}_O^I \rangle$$

To illustrate that the lightweight structural provenance stores less redundant information than the full-blown structural provenance from the previous section, we demonstrate it on our running example.

Example 8

We apply the lightweight structural provenance on the flatten operator in the example pipeline $F_{address1}^I$. It is the same operator that we used to illustrate the full-blown structural provenance in Example 7. We re-use the shortened input

in Table 5.1 from our previous example. We further assume that the input relation has the identifier 0 and the flatten operator's output relation has the identifier 1. Then, the lightweight schema extension is:

$$\mathcal{LSP}_{F^I_{address1}}^S = \left(\begin{array}{l} \text{oR} : 1, \\ \text{otype} : F^I, \\ \mathcal{I}^S : \left\{ \left\{ \left\langle iR : 0, \mathcal{A}^S : \left\{ \left\{ \langle p : address1[id].city \rangle, \right\} \right\} \right\} \right\} \right\}, \\ \mathcal{M}^S : \left\{ \left\{ \langle in : address1[id].city, out : city \rangle, \right\} \right\} \\ \left\{ \left\{ \langle in : address1[id].year, out : year \rangle \right\} \right\} \end{array} \right)$$

Each line in the tuple brackets holds one of $\mathcal{LSP}_{F^I_{address1}}^S$'s attributes. The first line has the output relation 1, the second line the operator type F^I . The third line keeps the input provenance \mathcal{I}^S , which refers to the input relation and holds the access provenance \mathcal{A}^S . Its paths are tuple-independent as indicated by the identifier placeholder *id*. Similarly, the manipulation provenance \mathcal{M}^S in the last line has a tuple-independent mapping between the input and output paths. The lightweight instance extension $\mathcal{LSP}_{F^I_{address1}}^I$ holds the identifiers needed to replace the identifier placeholders *id* in $\mathcal{LSP}_{F^I_{address1}}^S$. As defined in Table 5.3 the instance extension holds a relation of tuples with three attributes. The input tuple with identifier *in* and the nested tuple with identifier *nest* contribute to the output tuple *out*:

$$\mathcal{LSP}_{F^I_{address1}}^I = \left(\left(\left(\langle in : 2, nest : 31, out : 113 \rangle, \right) \right) \right) \\ \left(\left(\langle in : 2, nest : 32, out : 114 \rangle, \right) \right) \\ \left(\left(\langle in : 3, nest : 51, out : 115 \rangle, \right) \right) \\ \left(\left(\langle in : 3, nest : 52, out : 116 \rangle \right) \right)$$

The instance extension $\mathcal{LSP}_{F^I_{address1}}^I$ holds one record for each output tuple. When the path identifier placeholders *id* in $\mathcal{LSP}_{F^I_{address1}}^S$ are substituted with the identifiers in $\mathcal{LSP}_{F^I_{address1}}^I$, we obtain exactly the same paths as the ones

in Table 5.2, which holds the “uncompressed” structural provenance next to the operator output.

Together, they form the lightweight provenance capture $\mathcal{LSP}_{F_{address1}^I}$:

$$\mathcal{LSP}_{F_{address1}^I} := \langle \mathcal{LSP}^S : \mathcal{LSP}_{F_{address1}^I}^S, \mathcal{LSP}^I : \mathcal{LSP}_{F_{address1}^I}^I \rangle$$

The example has illustrated that it is possible to recover the “uncompressed” structural provenance from the lightweight schema and instance extensions. Generally, it is easy to see that the lightweight schema and instance extensions together suffice to recover the “uncompressed” structural provenance for all defined operators. Next, we introduce the Pebble algorithm that captures the lightweight provenance introduced in this section to compute explanations for existing answers.

5.3 Computing explanations for existing data

Recall from Section 3.3 that the Pebble algorithm computes explanations for existing data in the result. It has a capture and a query phase. In our architecture overview Figure 3.3, we have illustrated the capture phase with bright-green arrows and the query phase with dark-green arrows. In this section, we briefly summarize how Pebble captures the lightweight provenance in the following paragraph. Then, we put the focus on how Pebble computes the explanations based on the captured provenance.

Pebble captures the lightweight provenance during the actual query execution. It stores the captured provenance in dedicated nested relations in the big data analytics system to query them once an explanation is requested.

5.3.1 Backtracing

Pebble computes the explanations for existing answers in its backtracing procedure. For that purpose, it needs pointers to the result data, for which an explanation is requested in addition to the lightweight provenance. It stores these pointers in so-called provenance trees. To obtain the pointers,

Pebble requires a tree-pattern as input parameter to the backtracing phase. It utilizes the tree-pattern matching algorithm introduced in Chapter 4 to obtain the paths in the result data that match the tree-pattern. Recall that the third variant of the data matching algorithm in Section 4.3.2 returns exactly the needed paths. Pebble computes a so-called provenance tree from these paths before it starts the backtracing procedure. The backtracing procedure moves backward from the output relation to the input relations operator by operator. This section illustrates the complete backtracing procedure starting with the tree-pattern.

Definition 5.9 (Provenance tree)

The provenance tree $\mathcal{PT} = \langle root, N \rangle$ has a root node $root$ and a node collection N . The root node always refers to a top-level tuple of a relation. Each node $n \in N$ is a 6-tuple:

$$n = \langle name : name_n, parent : parent_n, C : C_n, A : A_n, M : M_n, c : c_n \rangle$$

The name attribute matches an attribute name or represents a tuple identifier if its parent $parent$ has nested relation type. Further, n references its children C . These attributes define the structure of the provenance tree. To maintain the access and manipulation provenance during backtracing n further holds a set of operators A that access the referenced attribute and a set of operators M that manipulate the attribute. The boolean value c indicates whether the node contributes to the data ($c = true$) or whether it influences the data ($c = false$), for which an explanation is queried.

Recall that contributing data is all input data that produce the output data. Influencing data is all input data that does not directly produce the output data. However, it is accessed during query evaluation. Therefore, it has influenced the existence of the output data.

The provenance tree \mathcal{PT} is only defined for top-level tuples in a relation. Hence, it needs context. The backtracing structure provides this context because it holds tuple identifiers and a relation identifier.

Definition 5.10 (Backtracing structure)

When R is a relation, the provenance structure holds a provenance tree \mathcal{PT}_t for each $t \in R$:

$$\mathcal{B} = \langle rR : id(R), rT : \{\{\langle rt : id(t), \mathcal{PT} : \mathcal{PT}_t \rangle\}\} \rangle$$

The attribute rR references a relation by identifier and the attribute rT holds a nested relation of tuple identifiers in rt and provenance trees \mathcal{PT} rooted at t .

To illustrate the provenance tree and the backtracing structure, we provide an example tree and backtracing structure based on our running example.

city		nList	
101	LA	firstname	
		711	Sue
		712	Sue
		713	Tom

Table 5.4: Unexpected data that is existing in the example output (Table 1.3)

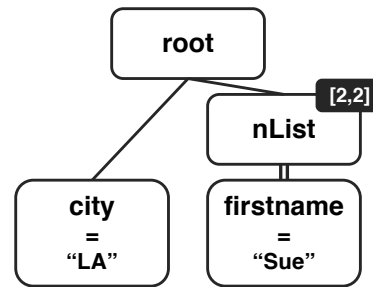


Figure 5.1: Example tree-pattern T (Figure 1.2a)

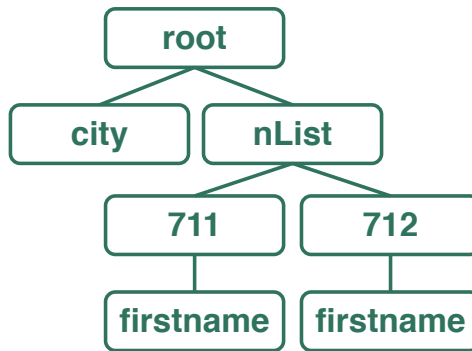


Figure 5.2: Example provenance tree \mathcal{PT}_{ex} obtained from matching the tree-pattern in Figure 5.1 on tuple 101 in Table 5.4

Example 9

When the tree-pattern matching algorithm applies the tree-pattern in Figure 5.1 on tuple 101 in the example output in Table 5.4, it yields the provenance tree \mathcal{PT}_{ex} shown in Figure 5.2. The green nodes in that provenance tree correspond

to the green fields in Table 5.4. Given that the relation in Table 5.4 has identifier 4, the complete backtracing structure is:

$$\mathcal{B}_4 = \langle rR : 4, rT : \{ \{ \langle rt : 101, \mathcal{PT} : \mathcal{PT}_{ex} \rangle \} \} \rangle$$

Pebble updates the content of the backtracing structure while stepping backward operator by operator. Before the update, the backtracing structure corresponds to the output of an operator. After the update, the backtracing structure corresponds to the input of the same operator. Hence, Pebble modifies the relation and tuple identifiers as well as the provenance trees to pinpoint the input data that precisely explain the existence of the selected data in the initially provided tree-pattern.

Pebble utilizes two helper methods to manipulate the provenance trees during backtracing. Their execution context is the lightweight provenance capture \mathcal{LSP} and the backtracing structure \mathcal{B} .

The *manipulatePath* method performs two tasks to track any data and structure manipulations. First, it manipulates the nodes in \mathcal{B} 's provenance trees \mathcal{PT} . For each path mapping that exists in \mathcal{PT} , it finds the corresponding path in the manipulation provenance \mathcal{M} . More precisely, it matches an output path *out* in \mathcal{M} and transforms it to the corresponding input path *in* in \mathcal{M} . After the transformations, the nodes in the tree \mathcal{PT} conform to the schema of the input relation. Second, it adds the current relation identifier rR from \mathcal{B} to each node's manipulation relation M .

The *accessPath* method adds the accessed attributes in the data to the nodes of the provenance tree \mathcal{PT} . It iterates over all accessed paths recorded in \mathcal{LSP} . For each path, it marks the visited nodes in the \mathcal{PT} . During the iteration, either of two cases occurs. In the first case, all nodes of the path already exist in \mathcal{PT} . Then, the method adds the current relation identifier rR from \mathcal{B} to each node's access relation A . In the second case, nodes that occur in the access path do not exist in the provenance tree \mathcal{PT} . This is possible because the accessed attributes are neither needed to reproduce the result nor have been accessed by other operators in the query. Then, the *accessPath* method adds the missing nodes to \mathcal{PT} and sets their contribution

value to $c = false$ since the nodes do not directly contribute to the requested data.

Given the backtracing structure with the provenance trees obtained from the tree-pattern matching algorithm and the helper methods *manipulatePath* and *accessPath*, Pebble starts the backtracing with Algorithm 6. The procedure recursively traces the data in the backtracing structure \mathcal{B} back from the query's output relation to its input relations. Its input is the lightweight provenance capture \mathcal{LSP} and the backtracing structure \mathcal{B} . Based on the operator type in \mathcal{LSP} , the algorithm recursively calls an operator-dependent backtracing procedure that returns its input's lightweight provenance capture \mathcal{LSP}' and an updated backtracing structure \mathcal{B}' . The recursion terminates when \mathcal{LSP}' is not defined. Then, the procedure has reached an input relation. In the following, we introduce the operator-dependent backtracing procedures.

Table access, projection, renaming, selection, tuple flatten, tuple nesting. The operators that have a single input relation and yield at most one top-level output tuple for each top-level input tuple share a generic backtracing procedure shown in Algorithm 7.

The algorithm has two major steps in addition to initialization (ll. 1-2) and finalization (ll. 9-11). During the initialization, the algorithm obtains the operator identifier oid and the input provenance \mathcal{I}_1^S . This is the only tuple in $\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{I}^S$ because the operators have exactly one input relation.

In the first step (l. 3), the backtracing algorithm joins the tuple identifiers in the backtracing structure $\mathcal{B}.rT$ with the output tuple identifiers in the lightweight provenance $\mathcal{LSP}.\mathcal{LSP}^I$. As a consequence, the input tuple identifiers in the lightweight provenance become the updated backtracing tuple identifiers in rT' . This join is essentially the same join that existing lineage solutions [IPW11; IST+15; LDY13] apply for backtracing flat data.

In the second step (ll. 4-8), the algorithm iterates over all the tuples t in rT' to update the provenance trees $t.\mathcal{PT}$. It undoes all recorded structural manipulations in the *manipulatePath* method (l. 6). Further, it marks the nodes in $t.\mathcal{PT}$ with the oid to indicate manipulations. The algorithm also

Algorithm 6: $\text{backtrace}(\mathcal{LSP}, \mathcal{B})$

Input: $\mathcal{LSP}, \mathcal{B}$ **Output:** $\mathcal{LSP}', \mathcal{B}'$

```
1 switch  $\mathcal{LSP}.\mathcal{LSP}^S.otype$  do
2   case  $\pi \vee \rho \vee \sigma \vee F^T \vee \mathcal{N}^T$  do
3      $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceGeneric}(\mathcal{LSP}, \mathcal{B})$ 
4   case  $\mathcal{N}^R$  do
5      $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceRelationNesting}(\mathcal{LSP}, \mathcal{B})$ 
6   case  $F^I$  do
7      $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceRelationFlatten}(\mathcal{LSP}, \mathcal{B})$ 
8   case  $\gamma$  do
9      $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceAggregation}(\mathcal{LSP}, \mathcal{B})$ 
10  case  $\bowtie$  do
11     $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceJoin}(\mathcal{LSP}, \mathcal{B}, dir)$ 
12  case  $\cup$  do
13     $\langle \mathcal{LSP}', \mathcal{B}' \rangle \leftarrow \text{backtraceUnion}(\mathcal{LSP}, \mathcal{B}, dir)$ 
14 if  $\mathcal{LSP}'$  is defined then
15    $\text{backtrace}(\langle \mathcal{LSP}', \mathcal{B}' \rangle)$ 
16 return  $\langle \mathcal{LSP}', \mathcal{B}' \rangle$ 
```

records the attribute access in $t.\mathcal{PT}$ by calling accessPath (l. 8) with the oid .

Once all paths in rT' are updated, finalization starts. The algorithm packages an updated backtracing structure \mathcal{B}' with rT' and the input's relation identifier $\mathcal{I}_1^S.iR$. It obtains the input's lightweight provenance \mathcal{LSP}' from the big data analytics system that is captured and stored during the initial query execution and returns a tuple with \mathcal{LSP}' and \mathcal{B}' (ll. 9-11).

Relation Nesting. The relation nesting creates nested relations as described in Section 5.1. Therefore, the relation nesting's lightweight provenance holds a collection of input tuples $inRel$ for each top-level output tuple

Algorithm 7: $\text{backtraceGeneric}(\mathcal{LSP}, \mathcal{B})$

Input: $\mathcal{LSP}, \mathcal{B}$ **Output:** $\langle \mathcal{LSP}', \mathcal{B}' \rangle$

```
1  $oid \leftarrow \mathcal{LSP}.\mathcal{LSP}^S.oR$ 
2  $\mathcal{I}_1^S \leftarrow \text{first}(\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{I}^S)$ 
3  $rT' \leftarrow \llbracket \rho_{in \leftarrow rt}(\pi_{in, \mathcal{PT}}(\mathcal{B}.rT \bowtie_{rt=out} \mathcal{LSP}.\mathcal{LSP}^I)) \rrbracket$ 
4 for  $t \in rT'$  do
5   for  $m \in \mathcal{LSP}.\mathcal{LSP}^S.\mathcal{M}^S$  do
6      $\text{manipulatePath}(t.\mathcal{PT}, m, oid)$ 
7   for  $a \in \mathcal{I}_1^S.\mathcal{A}^S$  do
8      $\text{accessPath}(t.\mathcal{PT}, a, oid)$ 
9  $\mathcal{B}' \leftarrow \langle rR : \mathcal{I}_1^S.oR, rT : rT' \rangle$ 
10  $\mathcal{LSP}' \leftarrow \text{get}(\mathcal{I}_1^S.iR)$ 
11 return  $\langle \mathcal{LSP}', \mathcal{B}' \rangle$ 
```

out (cf. Section 5.2). Given the lightweight provenance \mathcal{LSP} and the backtracing structure \mathcal{B} , Algorithm 8 traces back the relation nesting operator. Algorithm 8 has the same initialization and finalization routine (ll. 1-2 and 9-11) as the generic backtracing algorithm. Thus, we do not describe them again. The algorithm also has the two steps that join the backtracing structure with the lightweight provenance and update the provenance trees. However, the internals of these steps distinguish from the internals in the generic backtracing algorithm.

Before the algorithm conducts the same join as the generic algorithm (l. 4) to obtain rT' , it flattens the nested relation $inRel$ in $\mathcal{LSP}.\mathcal{LSP}^I$ into $\mathcal{LSP}^{I'}$ (l. 3). Each entry in the flattened $\mathcal{LSP}^{I'}$ matches one top-level input tuple. After the join, the algorithm adds a column $inProv$ to rT' which defaults to `false` (l. 5). It needs the additional column later to remove entries from rT' that are not part of the structural provenance.

The algorithm iterates over all tuples t in rT' (ll. 6-16) to update the provenance trees $t.\mathcal{PT}$ and to mark attribute access and manipulation. For

Algorithm 8: $\text{backtraceRelationNesting}(\mathcal{LSP}, \mathcal{B})$

Input: $\mathcal{LSP}, \mathcal{B}$ **Output:** $\langle \mathcal{LSP}', \mathcal{B}' \rangle$

```
1  $oid \leftarrow \mathcal{LSP}.\mathcal{LSP}^S.oR$ 
2  $\mathcal{I}_1^S \leftarrow \text{first}(\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{I}^S)$ 
3  $\mathcal{LSP}^{I'} \leftarrow \llbracket F_{inRel}^I(\mathcal{LSP}.\mathcal{LSP}^{I'}) \rrbracket$ 
4  $rT' \leftarrow \llbracket \rho_{in \leftarrow rt}(\pi_{in, \mathcal{PT}}(\mathcal{B}.rT \bowtie_{rt=out} \mathcal{LSP}^{I'})) \rrbracket$ 
5  $rT' \leftarrow \llbracket rT' \bowtie_{true} \{ \{ \langle inProv : false \rangle \} \} \rrbracket$ 
6 for  $t \in rT'$  do
7   for  $m \in \mathcal{LSP}.\mathcal{LSP}^S.\mathcal{M}^S$  do
8      $out \leftarrow m.out$ 
9     if  $out$  contains  $id$  then
10        $out \leftarrow \text{replace } id \text{ with } t.in \text{ in } out$ 
11     if  $out \in t.\mathcal{PT}$  then
12        $t.inProv = true$ 
13        $\text{manipulatePath}(t.\mathcal{PT}, \langle in : m.in, out : out \rangle, oid)$ 
14        $\text{removeNodes}(t.\mathcal{PT}, m.out)$ 
15   for  $a \in \mathcal{I}_1^S.\mathcal{A}^S$  do
16      $\text{accessPath}(t.\mathcal{PT}, a, oid)$ 
17  $rT' \leftarrow \llbracket \pi_{rt, \mathcal{PT}}(\sigma_{inProv=true}(rT')) \rrbracket$ 
18  $\mathcal{B}' \leftarrow \langle rR : \mathcal{I}_1^S.oR, rT : rT' \rangle$ 
19  $\mathcal{LSP}' \leftarrow \text{get}(\mathcal{I}_1^S.oR)$ 
20 return  $\langle \mathcal{LSP}', \mathcal{B}' \rangle$ 
```

each t , it iterates over all paths m recorded in the manipulation provenance $\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{M}^S$ (ll. 7-14). Since the output path $m.out$ contains the identifier placeholder id , the algorithm replaces the placeholder with the tuple identifier $t.in$ (l. 10) and stores the path in the variable out . If the path out occurs in $t.\mathcal{PT}$, the algorithm sets $inProv$ to $true$ to indicate that t remains part of rT' . Further, the algorithm updates the provenance tree so that it complies with the schema of the input relation. That is why it calls the

manipulatePath method to undo the schema manipulations of the nesting operator. It also calls the *removeNodes* method with the unmodified output path *m.out* to remove all nodes in the provenance tree that refer to tuples in the nested relation the operator has created (ll. 11-14). As the final step in the loop over *rT'*, the algorithm records the accessed attributes (l. 16).

Before the finalization step starts, Algorithm 8 removes all tuples *t* from *rT'* whose *inProv* equals false. These are tuples that reside in the nested relation but are not part of the requested provenance. The following example illustrates the need for this filter.

Example 10

We apply Algorithm 8 on the nesting operator in the running example to illustrate its internals. We reuse the backtracing structure \mathcal{B}_4 from Example 9 to find the cause for the duplicate “Sue” in “LA”:

$$\mathcal{B}_4 = \langle rR : 4, rT : \{ \langle rt : 101, \mathcal{PT} : \mathcal{PT}_{ex} \rangle \} \rangle$$

The relation identifier is 4 and the nested relation backtracing relation *rT* contains exactly one tuple that references output tuple 101 in Table 5.4 and holds the provenance tree \mathcal{PT}_{ex} in Figure 5.2. Let the input lightweight provenance capture \mathcal{LSP}_4 contain the manipulation provenance:

$$\mathcal{M}_4^S : \{ \langle in : \text{firstname}, out : nList[id].\text{firstname} \rangle \}$$

and the instance provenance:

$$\mathcal{LSP}_4^I = \left\{ \left\langle inRel : \left\{ \left\langle in : 711 \right\rangle, \left\langle in : 712 \right\rangle, \left\langle in : 713 \right\rangle \right\}, out : 101 \right\rangle \right\}$$

After initialization, the algorithm flattens \mathcal{LSP}_4^I , joins it with *rT*, and adds the *inProv* column to *rT'* (ll. 3-5). The resulting *rT'* is:

rt	\mathcal{PT}	$inProv$
711	\mathcal{PT}_{ex}	<i>false</i>
712	\mathcal{PT}_{ex}	<i>false</i>
713	\mathcal{PT}_{ex}	<i>false</i>

Now, each top-level input tuple has its own record t in rT' . It holds the input tuple identifier in rt , a copy of the unmodified tree-pattern \mathcal{PT}_{ex} in \mathcal{PT} and a *false* value in the attribute $inProv$. Since the tree-pattern \mathcal{PT}_{ex} holds a node that refers to the $nList$ attribute in the output, Algorithm 8 updates the \mathcal{PT}_{ex} for each record t in rT' based on the record in \mathcal{M}_4^S (ll. 6-16).

For record 711, the algorithm replaces the identifier placeholder id in path $nList[id].firstname$ with 711 (l. 10). Since the path $nList[711].firstname$ is in \mathcal{PT}_{ex} as shown in Figure 5.3, the algorithm sets $inProv$ to *true* and replaces the path with the input path to $firstname$. After the replacement, the provenance tree still holds paths into the $nList$ relation, e.g, to the nested tuple 712. Thus, the algorithm removes node $nList$ and all its children from the tree by calling `removeNode` (ll. 11-14).

The algorithm repeats the loop over rT' for the tuples 712 and 713. It handles 712 in the way as 711 because both contribute to the explanation for the duplicate “Sue” in the query result. When the algorithm checks tuple 713, the condition in line 11 fails, since the path with 713 does not occur in \mathcal{PT}_{ex} . It initially referred to the $firstname$ “Tom”, for which we did not request an explanation. After the algorithm has checked each t in rT' , rT' contains:

rt	\mathcal{PT}	$inProv$
711	$\mathcal{PT}_{preNest}$	<i>true</i>
712	$\mathcal{PT}_{preNest}$	<i>true</i>
713	\mathcal{PT}_{ex}	<i>false</i>

where $\mathcal{PT}_{preNest}$ is shown in Figure 5.3. The algorithm filters out the record with $rt = 713$, since $inProv$ is *false* (l. 17). Now, rT' exclusively contains provenance information for the requested $firstname$ “Sue”. The algorithm returns an updated provenance structure \mathcal{B}' with the filtered rT' and the lightweight provenance of the nestings’ predecessor \mathcal{LSP}' .

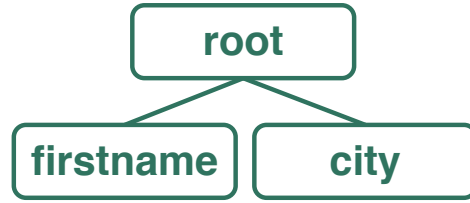


Figure 5.3: Example provenance tree $\mathcal{PT}_{preNest}$ obtained from Algorithm 8 for tuples 711 and 712

Relation flatten. To backtrack the relation flatten operator F^l , Algorithm 6 calls the *backtraceRelationFlatten* procedure in Algorithm 9. It resembles the generic backtracing procedure in Algorithm 7 because it has roughly the same two main steps wrapped in an initialization and finalization step.

After initialization (ll.1-2), the algorithm applies the same join as the generic algorithm (l. 3) but preserves the *nest* attribute in $\mathcal{LSP}.\mathcal{LSP}^l$. This attribute holds the tuple identifier in the nested relation that is flattened out.

Then, the algorithm updates the provenance trees \mathcal{PT} (ll. 4-8) like the generic Algorithm 7. Since the input paths *in* in the manipulation provenance $\mathcal{LSP}.\mathcal{LSP}^s.\mathcal{M}^s$ contain an identifier placeholder *id*, nodes in the provenance trees may hold the placeholder after the update. Further, recall that the flatten operator generally flattens one top-level input tuple into multiple top-level output tuples. Algorithm 9 considers this one-to-many relationship between input and output tuples. It has to merge multiple provenance trees from the output tuples into one provenance tree for each input tuple.

Therefore, the *backtraceRelationFlatten* procedure nests the provenance tree \mathcal{PT} and the identifier of the nested tuple in attribute *nest* into a nested relation X' for each distinct tuple identifier *rt* in rT' (l. 9). Then, it applies the *mergeTrees* aggregate function on X' to merge the trees under each *rt* into a single tree. It moves attributes the flatten operator unnests under a single node that represents the nested collection that the operator has

Algorithm 9: $\text{backtraceRelationFlatten}(\mathcal{P}, \mathcal{B})$

Input: $\mathcal{LSP}, \mathcal{B}$ **Output:** $\langle \mathcal{LSP}', \mathcal{B}' \rangle$

```
1  $oid \leftarrow \mathcal{LSP}.\mathcal{LSP}^S.oR$ 
2  $\mathcal{I}_1^S \leftarrow \text{first}(\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{I}^S)$ 
3  $rT' \leftarrow \llbracket \rho_{in \leftarrow rt}(\pi_{in, nest, \mathcal{PT}}(\mathcal{B}.rT \bowtie_{rt=out} \mathcal{LSP}.\mathcal{LSP}^I)) \rrbracket$ 
4 for  $t \in rT'$  do
5   for  $m \in \mathcal{LSP}.\mathcal{LSP}^S.\mathcal{M}^S$  do
6      $\text{manipulatePath}(t.\mathcal{PT}, m, oid)$ 
7   for  $a \in \mathcal{I}_1^S.\mathcal{A}^S$  do
8      $\text{accessPath}(t.\mathcal{PT}, a, oid)$ 
9  $rT' \leftarrow \llbracket \gamma_{\text{mergeTrees}(X') \rightarrow \mathcal{PT}}(\mathcal{N}_{X \rightarrow X'}^R(\mathcal{N}_{\{nest, \mathcal{PT}\} \rightarrow X}^T(rT')))) \rrbracket$ 
10  $\mathcal{B}' \leftarrow \langle rR : \mathcal{I}_1^S.oR, rT : rT' \rangle$ 
11  $\mathcal{LSP}' \leftarrow \text{get}(\mathcal{I}_1^S.iR)$ 
12 return  $\langle \mathcal{LSP}', \mathcal{B}' \rangle$ 
```

flattened out. During that process, the function replaces the placeholder id in the merged trees with the actual values in $nest$.

Eventually, the algorithm conducts the same finalization steps as the backtraceGeneric algorithm to obtain and return \mathcal{B}' and \mathcal{LSP}' .

Example 11

To illustrate the backtracing of the relation nesting operator, we apply Algorithm 9 on our running example. Recall that we like to get an explanation for the duplicate firstname “Sue” in the nested list of the city “LA”. Example 8 has shown the lightweight provenance capture for the flatten operator $\mathcal{LSP}_{\text{address1}}^I$. When the algorithm has applied the generic backtrace function in Algorithm 7 to backtrace the projection and filter operator in the query pipeline of Figure 1.1, it has computed the backtracing structure \mathcal{B}_1 that contains the following rT :

rt	\mathcal{PT}
113	$\mathcal{PT}_{\text{postFlatten}}$
116	$\mathcal{PT}_{\text{postFlatten}}$

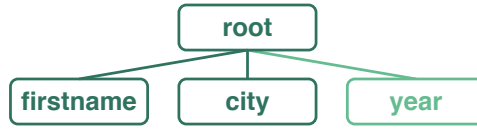


Figure 5.4: Example provenance tree $\mathcal{PT}_{postFlatten}$ obtained after backtracing the pipeline in Figure 1.1 to relation flatten operator for tuples 113 and 116

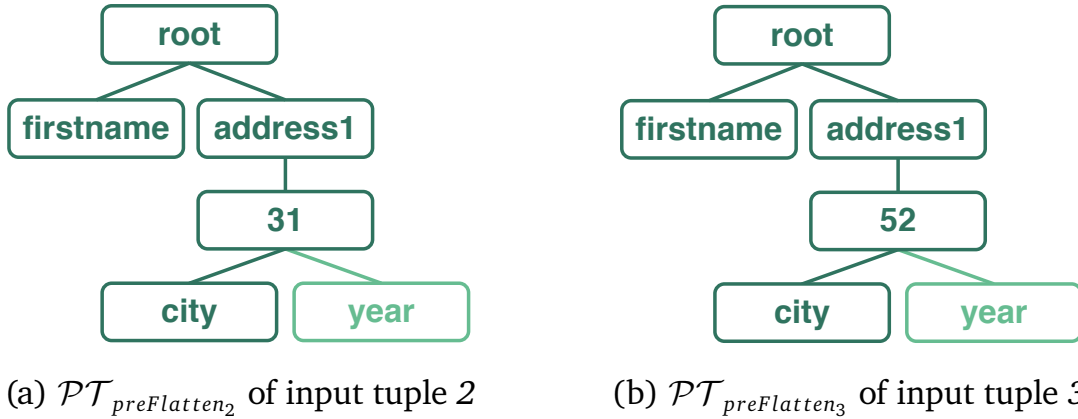


Figure 5.5: Provenance trees \mathcal{PT} on the input relation, which yield exactly the highlighted cells in Table 5.5

Figure 5.4 shows the provenance tree $\mathcal{PT}_{postFlatten}$. In addition to the three nodes in the tree of Figure 5.4, the provenance tree $\mathcal{PT}_{postFlatten}$ has a light green year node since the selection in the example pipeline accesses the year and the backtracing algorithm marks it as an influencing attribute.

Algorithm 9 joins the $\mathcal{LSP}_{F^I_{address1}}$ with rT (l. 3) and obtains rT' :

rt	$nest$	\mathcal{PT}
2	31	$\mathcal{PT}_{postFlatten}$
3	52	$\mathcal{PT}_{postFlatten}$

Then, it iterates over the two records in rT' to replace the paths to year and city with $address1[id].year$ and $address1[id].city$, respectively (l. 6). In our example, the two records in rT' have different values in rt . Thus, the algorithm does not merge them into a single record, when it merges the trees (l. 9). However, it replaces the placeholder id in $address1[id]...$ with the

	firstname	lastname	address1		address2	
1	Peter	Jones	city	year	city	year
			LA	2010	LA	2019
			SF	2018	LV	2018
					SF	2017
2	Sue	Miller	city	year	city	year
			LA	2019	NY	2018
			NY	2018	LA	2019
3	Sue	Walker	city	year	city	year
			SF	2018	LV	2018
			LA	2019	NY	2019
4	Tom	Smith	city	year		
			LA	2019		

Table 5.5: Explanations for the **existing** example output, with **contributing** and **influencing** data (Table 1.5)

identifiers in nest. The resulting trees are shown in Figure 5.5. Eventually the algorithm returns a backtracing structure \mathcal{B}' that holds the following rT' :

$$\begin{array}{r}
 rt \quad \mathcal{PT} \\
 \hline
 2 \quad \mathcal{PT}_{preFlatten_2} \\
 3 \quad \mathcal{PT}_{preFlatten_3}
 \end{array}$$

When this rT' is applied to the input data, we obtain exactly the highlighted cells in Table 5.5. Note, that this step does not require any tree-pattern matching, since the paths in the provenance trees directly point to the highlighted values in Table 5.5.

Even though the algorithm has reached the input relation in the running example, we have not described the backtracing procedure for the aggregation, join, and union operator. We sketch the algorithms for these operators below.

Aggregation. The backtracing procedure for the aggregation operator resembles the generic backtracing procedure in Algorithm 7, for the ag-

gregation has one input relation and creates exactly one output tuple for each input tuple. However, it differs from the generic backtracing procedure in updating the manipulated paths. Recall that the aggregation computes an aggregated value from a nested relation. Thus, for each manipulation path in $\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{M}^S$ that contains an identifier placeholder id , the algorithm iterates over all records t' in the attribute $nRel$ that is part of the lightweight instance provenance \mathcal{LSP}_γ^I . It replaces the placeholder with the value in $t'.nest$ and updates the provenance tree accordingly. The rest of the procedure matches the generic procedure.

Join and union. Unlike the other operators, the join and the union operator have two input relations. Hence, the backtracing procedures require an additional direction parameter dir that specifies the input relation to which the algorithm traces the provenance back. Based on that parameter, the algorithms choose the correct input provenance in the lightweight schema provenance $\mathcal{LSP}.\mathcal{LSP}^S.\mathcal{I}$ and choose the correct attribute lin or rin in the lightweight instance provenance $\mathcal{LSP}.\mathcal{LSP}^I$ (cf. Table 5.3). Then, they call the generic backtracing procedure from Algorithm 7. Afterward, the join algorithm removes all nodes in the provenance trees \mathcal{PT} that are not part of the chosen input schema since they reference elements in the schema of the other input by calling the *removeNodes* method known from Algorithm 8. The algorithm for the union operator filters out all items in rT' whose value is undefined in the chosen attribute lin or rin . These items originate from the other input relation of the union operator.

Now, we have introduced all backtracing procedures. Hence, we move on to a discussion on the algorithm's runtime complexity and capability to scale to large datasets in big data analytics systems.

5.3.2 Discussion

The Pebble algorithm has a dedicated provenance capture and query phase. Here, we describe the runtime complexity of these phases and explain how these phases scale to large datasets in big data analytics systems.

Runtime complexity. The lightweight provenance capture computes unique identifier annotations to annotate top-level tuples. Pebble can compute these annotations in constant time and with constant space for each tuple. Thus, the runtime complexity for lightweight provenance capture is $O(d)$ for each operator in the query, where d is the number of top-level tuples in the output relation of that operator. The only exception is the relation nesting operator that collects a nested relation of input identifiers for each output tuple. It has a complexity of $O(i)$, where i is the number of top-level input tuples of the relation nesting operator. Note that all operators have a minimum runtime complexity of $O(\max(d, i))$. Hence, the lightweight provenance capture does not have an impact on the theoretical query complexity. However, computing the annotations takes time in practice. Therefore, measurable runtime overhead exists as we will show in Chapter 8.

Computing the explanations based on the lightweight provenance annotations first requires applying the tree-pattern matching algorithm from Chapter 4. Its runtime complexity is described in Section 4.3.3. Furthermore, Pebble conducts one join operation for each operator in the query. When Pebble applies hash-join, each join's runtime complexity is approximated as $O(\max(d, i))$. Note that the number of input tuples i and output tuples d varies within a query and depends on the operator type. For each operator, Pebble further conducts a number of path updates p on each top-level tuple. The number of path updates is bound by the number of data values in the top-level tuples. Thus, the explanation computation has the per-operator complexity of $O(\max(d, i) \cdot p)$. Given that the query pipeline has q operators, the resulting complexity is $O(\max(d, i) \cdot p \cdot q)$. While the theoretical complexity is feasible, the explanation computation is computationally intensive in practice.

As we will discuss in detail in Chapter 8, the size of the intermediate results and the join operations are mainly responsible for the high practical overhead. Recall that Pebble collects each operator's annotations in separate nested relations during query execution. When computing the explanations, Pebble has to load all annotations for each operator from the big data analytics

system. The explanations may reference only a small subset of the processed data. Hence, Pebble potentially loads large amounts of data without needing it. Furthermore, the backtracing requires one join for every operator in the query, even for selections and projections. The latter two operators do not require shuffling the data across computing nodes. However, the join operations require shuffling the data. That involves time-consuming network I/O, which lets the runtime further increase compared to the original query. Nonetheless, Pebble is capable of providing explanations on large datasets in distributed computing environments as we discuss next.

Scalability. In the introduction (Contribution (3)), we have stated that we contribute the novel Pebble algorithm to compute explanations for existing data in the result. We claim that it scales to large datasets. Here, we discuss how Pebble’s execution can be distributed on big data analytics systems. The lightweight provenance capture phase exclusively computes annotations for individual top-level tuples. Since computing the annotations of one top-level tuple is independent of computing the annotations for all other top-level tuples, this phase is trivially parallelizable. Hence, the provenance capture phase scales to large datasets on big data analytics systems.

Computing the explanations in Pebble’s second phase mainly consists of joins and the manipulation of the provenance trees. The big data analytics systems provide the means to execute joins in a distributed and scalable fashion on compute clusters. Hence, Pebble scales towards that end. Recall that provenance trees correspond to exactly one top-level tuple in the processed relation. Thus, manipulating different trees simultaneously is trivial. Therefore, computing the explanations also scales on distributed big data analytics systems.

5.4 Summary

In this chapter, we have described our research Contribution (3) in detail. We have defined the formal provenance capture semantics for our novel structural provenance to faithfully capture provenance for nested data. Structural

provenance yields more comprehensive explanations than other existing provenance since they provide information on the access and manipulation of the data at the granularity of individual attributes.

However, directly capturing the provenance according to the formal provenance capture semantics implies collecting large amounts of redundant information. Therefore, we introduce the lightweight structural provenance that avoids collecting redundant information by splitting the provenance into a schema-based component and an instance-based component.

Our novel Pebble algorithm captures this lightweight provenance to compute explanations on request for select nested data. During the explanation computation, Pebble resolves structural dependencies and data dependencies at the granularity of individual nested attributes. The computation is designed to scale to large datasets on big data analytics systems. Therefore, Pebble is the first algorithm that provides explanations at the granularity of individual nested attributes and, at the same time, scales to large datasets.

EXPLANATIONS FOR MISSING DATA

Complex analytical queries process nested data in big data analytics systems. If they yield a result that misses expected data, the systems provide no built-in means to analyze the root cause of the missing data. We provide the first solution that finds query-based explanations for missing data in the result of big data analytics queries over nested data. Related existing solutions are limited to flat relational data and do not scale to large datasets. Furthermore, their query-based explanations exclusively rely on provenance. Hence, their explanations only contain selective operators, which remove data that potentially contribute to the missing result.

In this chapter, we propose a novel approach to query-based explanations that overcomes the three mentioned shortcomings. It extends the provenance-based approach with reparameterizations. Reparameterizations modify operator parameters such that the missing answer appears in the result. Hence, our novel approach potentially contains any parameterizable operator type in the explanations. It even accounts for misinterpreted attributes

referenced in the analytical query. Therefore, it even finds explanations on flat data that existing solutions cannot find. Further, it is the first approach to support nested data and to scale to practically meaningful dataset sizes on distributed big data analytics systems.

This chapter describes Contribution (4) in detail and is based on four published papers [DGHL19; DLGH21; DLHG21a; DLHG21b]. The topics discussed in this work focus on those aspects in the papers that I have contributed to them. In this chapter, we formally introduce the why-not question in Section 6.1. It describes the missing data in the result of a big data analytics query. Then, in Section 6.2, we define reparameterizations based on the execution model in Table 3.1 to set the bounds for allowable changes to the big data analytics query. Based on the reparameterizations, we introduce minimal successful reparameterizations. They let the missing data described in the why-not question appear in the result and keep the impact to query and result as low as possible to avoid unnecessary modifications to the query and prevent a blow-up of the query result. The reparameterized operators in the minimal successful reparameterizations become our query-based explanations for missing data in the result.

Even in large, distributed big data analytics systems, it is computationally infeasible to precisely compute these explanations. Hence, we propose the heuristic Breadcrumb algorithm in Section 6.3, which computes approximated explanations based on the reparameterizations. The Breadcrumb algorithm is one major module in our overall architecture in Section 3.3. We discuss in detail, how Breadcrumb computes the explanations in the dedicated section. We conclude the chapter with a summary of the contributions discussed in this chapter. Next, we introduce the why-not question.

6.1 Why-not question

Breadcrumb takes a why-not question as input. This section formally introduces the why-not question. In brief, the question describes a non-empty set of missing, yet expected top-level tuples in a query's result. For conve-

nience, the missing set may hold placeholders. The *instance placeholder* $?$ represents any value of any type. The *multiplicity placeholder* $*$ describes tuples in nested relations. They represent an arbitrary number of nested tuples, including 0. We call the of missing top-level tuples *nested instances with placeholders* (NIP).

Definition 6.1 (Nested instances with placeholders)

Let I be nested instance of the nested type τ as defined in our data model in Section 3.1, the instance placeholder $?$, or the multiplicity placeholder $*$. Then the rules to recursively construct nested instances with placeholders (NIPs) of type τ are:

- **Instance placeholder.** If $I = ?$, then I is a NIP of type τ .
- **Primitive type.** If $\text{type}(I) = \tau$, then I is a NIP of type τ .
- **Tuple type.** If $\tau = \langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle$, then $\langle I_1, \dots, I_n \rangle$ is a NIP of type τ if each I_i is a NIP of type τ_i .
- **Relation Type.** If $\tau = \{\{\tau_{tup}\}\}$, then $\{\{I_1, \dots, I_n\}\}$ is a NIP of type τ if
 - (i) $\forall I_i$ either $\text{type}(I_i) = \tau_{tup}$, $I_i = ?$, or $I_i = *$ and
 - (ii) $\nexists i \neq j \in \{1, \dots, n\}$ such that $I_i = I_j = *$

For finite domains, why-not questions with NIPs do not add expressive power to questions without them. However, efficiently supporting the former avoids an exponential blow-up when naively translating the former to the latter.

Next, we formally define the set of nested instances encoded by a NIP. Those are instances that match the NIP. To ensure that a why-not question asks for a tuple that is not part of the result, we require that none of the result tuples matches the why-not question’s NIP.

Definition 6.2 (Matching NIPs)

An instance I of type τ matches a NIP I' of type τ written as $I \simeq I'$ if one of these conditions holds:

- **Instance placeholder.** $I' = ?$

- **Primitive type.** $I = I'$
- **Tuple type.** $\text{type}(I) = \langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle$ and $\forall i \in [1, n], I.A_i \simeq I'.A_i$
- **Relation type.** $\text{type}(I) = \{\{\tau_{tup}\}\}$ and there exists an assignment $\mathcal{M} \subseteq I \times I' \rightarrow \mathbb{N}$ with the following properties:
 - (i) $\forall \langle t, t' \rangle \in \mathcal{M} : t \simeq t' \vee t' = ? \vee t' = *$
 - (ii) $\forall t \in I, t' \in I' : \mathcal{M}(t, t') > 0$
 - (iii) $\forall t \in I : \sum_{t' \in I'} \mathcal{M}(t, t') = \text{MULT}(I, t)$
 - (iv) $\forall t' \in I' : \sum_{t \in I} \mathcal{M}(t, t') = \text{MULT}(I', t') \vee t' = *$

The first three conditions cover the instance placeholder, primitive types, and tuple types. These conditions are self-explanatory. The last condition covers relation types. It ensures proper handling of bag semantics. For instance, consider a nested relation R with a single tuple t appearing three times ($\text{MULT}(R, t) = 3$). Then the NIP R' with $\text{MULT}(R', t) = 1$ and $\text{MULT}(R', ?) = 2$ matches R by assigning the two duplicates of $?$ to t .

To ensure that a why-not question asks for a tuple that is not part of the result, the NIP must not match any tuple in the query result. Then, the why-not question is the following triple.

Definition 6.3 (Why-not question)

Let Q be a query, D a database, and $\text{type}(\llbracket Q \rrbracket_D) = \{\{\tau\}\}$. A why-not question Φ is a triple $\Phi := (Q, D, t)$ where the why-not tuple t is a NIP of type τ that does not match any tuple in the result, i.e., $\forall t' \in \llbracket Q \rrbracket_D : t' \not\sim t$.

We illustrate the why-not question on our running example.

city	nList		
\emptyset	<table border="1"> <thead> <tr> <th>firstname</th> </tr> </thead> <tbody> <tr> <td>*</td> </tr> </tbody> </table>	firstname	*
firstname			
*			
NY			

Table 6.1: Expected data that is **missing** in the example output as introduced in Table 1.4

Example 12

In the running example, the query $Q_{example}$'s result lacks the city "NY". Therefore, we have informally described the missing tuple in Table 1.4, which we show again in Table 6.1 for convenience. The according NIP $t_{missing}$ is:

$$t_{missing} = \langle city : NY, nList : \{\{*\}\} \rangle$$

The tuple $t_{missing}$ requires the *city* attribute to hold the value "NY", while it allows for arbitrary number of tuples in the *nList* attribute, as indicated by the nested relation with the multiplicity placeholder $\{\{*\}\}$.

Given the NIP $t_{missing}$, the why-not question $\Phi_{example}$ is:

$$\Phi_{example} = \langle Q_{example}, D_{example}, t_{missing} \rangle$$

The why-not question $\Phi_{example}$ holds the example query $Q_{example}$ over database instance $D_{example}$ that holds the input relation in Table 1.1 and the NIP $t_{missing}$.

6.2 Query reparameterizations and explanations

This section describes our novel definition of query-based why-not explanations for a given why-not question. Our explanations hold sets of operators. These sets have the following three main characteristics:

- (i) The set of returned explanations pinpoints all possible combinations of operators that conjunctively cause tuples matching the NIP t of our why-not question to be missing from the query result.
- (ii) The set of returned explanations takes into account parameterizations of *all* query operators. Thus, each explanation may include any operator of Table 3.1 with parameters.
- (iii) The set of returned explanations only holds *minimal* explanations regarding the operators and the data in the query result. It considers changes to the original query result beyond the appearance of missing answers, called *side-effects*.

Existing lineage-based explanations lack all three characteristics. (i) They possibly yield incomplete explanations (false negatives) [BHT14; CJ09; Her15], when multiple operators require changes to make the missing data appear in the result. That shortcoming motivated alternative definitions [BHT15; DFGH18] which are limited to conjunctive queries. (ii) They are limited to explanations that contain only data-pruning operators (i.e. selections and joins). Thus, they miss causes at the schema level such as projecting on the wrong attribute. (iii) Finally, they disregard side-effects. They have only been considered for instance-based and refinement-based explanations as surveyed in [HDB17]).

Our explanations address all three characteristics for more complex queries than conjunctive queries and apply to flat and nested data. They are based on reparameterizations, which we introduce before the explanations.

Reparameterizations are modifications to the query in the why-not question that preserve the query structure and the result schema. Thus, they preserve operator types and dependencies between the operators but modify the operator parameters. We consider the following parameter changes admissible. This selection of changes is motivated by commonly arising errors in the field. Nonetheless, our formalism applies to alternative definitions of valid parameter changes as well.

Definition 6.4 (Valid parameter changes)

Given an operator O of a query Q , the operator’s parameters are defined as $param(Q, O)$. Their valid changes as summarized in Table 6.2.

To illustrate the content of Table 6.2, we discuss allowable changes to the selection in the following example.

Example 13

Recall that the selection in our running example filters on the year $\sigma_{year \geq 2019}$. According to Table 6.2, we can replace the year attribute with another attribute of the same type. Furthermore, we can replace the comparison operator \geq with any other operator $\{=, >, \geq, <, \leq, \neq\}$. Finally, we can substitute the constant value 2019 with any other value of the same type.

Operator op	Parameters $param(Q, op)$
Valid parameter changes	
Projection $\pi_{A_1, \dots, A_n}(R)$	$\{(A_1, \dots, A_n)\}$
Any substitution of an attribute A_i with an attribute A_j from R	
Renaming $\rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R)$	$\{(B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n)\}$
Changing the output attributes based on a permutation of (B_1, \dots, B_n)	
Selection $\sigma_\theta(R)$	$\{\theta\}$
(i) Replacing any reference to an attribute A in θ with another attribute B from R with the same data type; (ii) modifying comparison operators in $\{=, >, \geq, <, \leq, \neq\}$ to one another; and (iii) changing constant values to other constants of same type.	
Joins $op = R \diamond_\theta S$, where $\diamond \in \{\bowtie, \bowtie, \bowtie, \bowtie\}$	$\{\theta, type(op)\}$, where $type(op) = \diamond$
(i) Changing the join type of op ; (ii) replacing a reference to an attribute A with a different attribute B in θ ; (iii) modifying comparison operators in $\{=, >, \geq, <, \leq, \neq\}$ to one another.	
Tuple Flatten $F_A^T(R)$	$\{A\}$
Replacing A by an attribute B in R of tuple type	
Relation Flatten $F_A^I(R)$ or $F_A^O(R)$	$\{A, type(op)\}$, $type(op) = inner$ for inner flatten, $type(op) = outer$ for outer flatten
(i) Changing the attribute A to be flattened out and (ii) changing the flattening type	
Nesting $\mathcal{N}_{A \rightarrow C}^R(R)$ or $\mathcal{N}_{A \rightarrow C}^T(R)$	$\{A, C\}$
(i) Changing the attributes to be nested / grouped-on (A) or (ii) the name of the attribute storing the result of nesting (C)	
Aggregation $\gamma_{f(A) \rightarrow B}(R)$	$\{A, B, f\}$
(i) Changing the aggregation function f , (ii) the attribute that we are aggregating over (A), or (iii) the name of the attribute storing the aggregation result (B)	

Table 6.2: Valid parameter changes

The reparameterizations of a query Q are all those queries Q' that can be derived from Q through valid parameter changes.

Definition 6.5 (Reparameterizations (RPs))

Let Q be a query. The query Q' is a reparameterization of Q if a sequence of valid parameter changes exists that transforms Q into Q' .

For the ease of presentation, the operators O in Q retain their identifier in Q' after reparameterization. The reparameterizations are unrelated to a why-not question Φ . They do not indicate whether the reparameterization causes Q' to produce a result tuple that matches the why-not tuple in Φ . Therefore, we define successful reparameterizations, which are the reparameterizations that produce the missing data.

Definition 6.6 (Successful Reparameterizations (SRs))

Let $\Phi = (Q, D, t)$ be a why-not question and let $RE(Q)$ denote the set of all RPs for the query Q in Φ . Then, the set of successful reparameterizations with respect to Φ is:

$$SR(\Phi) = \{Q' \mid \exists t' \in \llbracket Q' \rrbracket_D, t' \simeq t \wedge Q' \in RE(Q)\}$$

While successful reparameterizations yield an explanation to the why-not question, they have two shortcomings. On the one hand, SRs may apply unnecessary changes to the Q . On the other hand, some SRs potentially cause more changes to the original query result than others.

Let O_1 and O_2 be operators in Q and let a parameter change in O_1 yield a successful reparameterization Q' . Further, let Q'' be derived from Q' such that it is still a successfully reparameterization. Furthermore, let the result of the two reparameterizations be the same, i.e., $\llbracket Q' \rrbracket = \llbracket Q'' \rrbracket$. Then, Q' and Q'' are both successful reparameterizations, but only the former SR precisely pinpoints the set of operators, i.e., $\{O_1\}$, that causes the missing data in the result. It is a true subset of the operators modified in Q'' , i.e., $\{O_1\} \subset \{O_1, O_2\}$. Extending this experiment to further operators in Q , shows that the set of reparameterized operators in the explanations should be minimal, in the sense that removing an operator from the successful reparameterization makes the reparameterization fail to produce the missing data.

A selection in a query Q may have two reparameterizations that both yield the missing data in the result, but one is more restrictive than the other. In this case, the former reparameterization may be favorable since it imposes less data changed in the result than the latter one. The data changed in the result that do not match the why-not tuple are side-effects. The goal is to minimize these side-effects.

Our goal is to keep the set of operators to a minimum and keep side-effects as low as possible. Since the two goals potentially conflict with each other, we define a partial order \preceq_Φ over SRs.

Definition 6.7 (Partial order over SRs \preceq_Φ)

Let Q be a query, D a database, $\Phi = (Q, D, t)$ a why-not question, and Q', Q'' be two SRs of Q . Let $\Delta(Q, Q')$ denote the set of operator identifiers whose parameters differ between Q and Q' , i.e., $\Delta(Q, Q') = \{O \mid \text{param}(Q, O) \neq \text{param}(Q', O)\}$. Further, let d be a distance function quantifying the distance between two nested relations. We define a partial order $Q' \preceq_\Phi Q''$ as follows:

- (i) $\Delta(Q, Q') \subseteq \Delta(Q, Q'')$
- (ii) $d(\llbracket Q \rrbracket_D, \llbracket Q' \rrbracket_D) \leq d(\llbracket Q \rrbracket_D, \llbracket Q'' \rrbracket_D)$

The definition of MSR provides an open choice on the distance function that measures the differences between two query results. To support nested and flat data equally well, we resort to the tree-edit distance for unsorted trees [Bil05; PA11], which computes the minimum cost of transforming one tree into another. It is restricted to the three edit operations: (i) node insertion, (ii) node deletion, and (iii) node update. Computing the tree edit distance of unordered trees is NP-hard [ZSS92]. We define the minimal successful reparameterizations as SRs that are minimal according to the partial order \preceq_Φ .

Definition 6.8 (Minimal successful reparameterizations (MSRs))

Let Q be a query, D a database, $\Phi = (Q, D, t)$ a why-not question. Further let Q', Q'' be two members of the set of all successful reparameterizations for the given why-not question $SR(\Phi)$. Then, Q' is a minimal successful reparameterization, if

$$\neg \exists Q'' \in SR(\Phi) : Q'' \preceq_\Phi Q'$$

Based on the minimal successful reparameterizations, we define explanations.

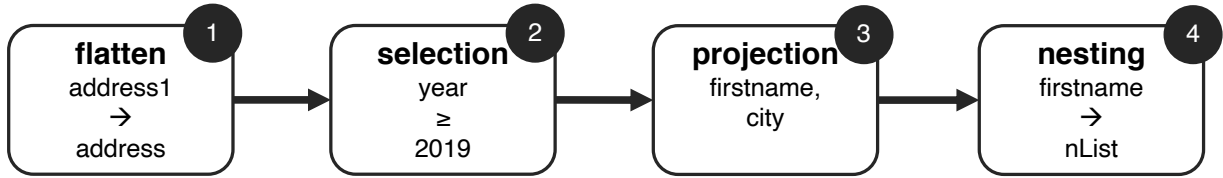


Figure 6.1: Example execution pipeline from Figure 1.1

Definition 6.9 (Explanations)

Let Φ be a why-not question and $MSR(\Phi)$ be the set of MSR for Φ . Then, the explanations $\mathcal{E}(\Phi)$ with respect to Φ are

$$\mathcal{E}(\Phi) := \{\Delta(Q, Q') \mid Q' \in MSR(\Phi)\}$$

According to the above definition, the explanations only contain the operators that are modified to obtain a minimal successful reparameterization. Next, we provide an example for reparameterizations, successful reparameterizations, minimal successful and reparameterizations, and explanations to illustrate their meaning.

Example 14

This example builds on the running example in Section 1.2. For convenience, we show the query pipeline $Q_{example}$ again in Figure 6.1.

While the number of possible reparameterizations is too large to be completely listed here, we pick four reparameterizations for the example query $Q_{example}$ to illustrate the concepts introduced in the above section:

- Q_0 : Change the flatten operator 1 from an inner flatten to an outer flatten.
- Q_1 : Replace the filter condition in operator 2 from $year \geq 2019$ to $year \geq 2018$.
- Q_2 : Flatten *address2* instead of *address1* in operator 1.
- Q_3 : Combine Q_1 and Q_2 .

Table 6.3 shows $Q_{example}$'s and Q_0 's results, which are the same in this example. Table 6.5 holds Q_1 's result, Table 6.4 exposes Q_2 's result, and Table 6.6 illustrates Q_3 's result. We introduce the meaning of the colors later in this example.

city	nList
LA	firstname
	Sue
	Sue Tom

Table 6.3: Result of $Q_{example}$ and the reparameterized query Q_0

city	nList
LA	firstname Peter
NY	firstname Sue Sue

Table 6.4: Result of the reparameterized query Q_2

city	nList
LA	firstname Sue Sue Tom
SF	firstname Peter Sue
NY	firstname Sue

Table 6.5: Result of the reparameterized query Q_1

city	nList
LA	firstname Peter
NY	firstname Sue Sue
LV	firstname Peter Sue

Table 6.6: Result of the reparameterized query Q_3

Next, we assess the reparameterizations Q_0 to Q_3 based on the example why-not question to identify successful reparameterizations. The example why-not question expects a tuple in the result that holds the value “NY” in its *city* attribute.

The reparameterized query Q_0 does not produce a tuple with the value “NY” in the attribute *city*. Thus, Q_0 is an RP, but not an $SR(\Phi_{example})$. Unlike Q_0 , the queries Q_1 , Q_2 , and Q_3 yield a result tuple that holds “NY” in the attribute *city* together with an associated bag of names in *nList*. The green fields in Table 6.5, Table 6.4, and Table 6.6 highlight these tuples. They are successful reparameterizations $SR(\Phi_{example})$.

We advance to checking whether the successful reparameterizations Q_1 , Q_2 , and Q_3 are minimal successful reparameterizations MSR. We start with Q_1 . It only modifies the selection operator. Thus, no reparameterization exists, that includes only a subset of the operators. However, it produces a red tuple in Table 6.5 as side-effect. Given the example query, the input data and the possible reparameterizations, this side-effect is minimal, because it changes the value 2019 to the next smaller value 2018 in the filter condition. No other modification of the filter condition yields the missing tuple and has less side-effects at the same time. Thus Q_1 is a minimal successful reparameterization. It yields a set containing the selection with identifier 2, i.e., $\{\sigma^2\}$, as explanation to the example why-not question.

Now let us have a closer look into the reparameterizations that involve the flatten operator. In general, we have to options to reparameterize the flatten. We can replace an inner flatten with an outer flatten as in Q_0 or replace the attribute in the flatten as in Q_2 . As discussed, Q_0 is no successful reparameterization, but Q_2 is. Hence, Q_2 also is a minimal successful reparameterization, because it only modifies the flatten operator. However, it has more side-effects than Q_1 since replaces the firstnames “Sue” (2x) and “Tom” with “Peter” in the “LA” tuple in Table 6.4.

Query Q_3 modifies the flatten and the selection operator. When comparing Q_3 to Q_2 , the condition $\Delta(Q_{example}, Q_2) \subseteq \Delta(Q_{example}, Q_3)$ holds. For, Q_2 only modifies the flatten operator. Comparing the result of Q_2 in Table 6.4 to the result of Q_3 in Table 6.6, further reveals that Q_3 has more side-effects than Q_2 . Thus, $d(\llbracket Q_{example} \rrbracket_D, \llbracket Q_2 \rrbracket_D) \leq d(\llbracket Q_3 \rrbracket_D, \llbracket Q_{example} \rrbracket_D)$ holds. Consequently, $Q_2 \preceq_{\Phi} Q_3$ holds and Q_3 is no minimal successful reparameterization.

We briefly discuss the complexity of computing the explanations according to the above definitions before we introduce the Breadcrumb algorithm. Recall that computing the tree-edit distance for unsorted trees is NP-hard. Even if we considered an alternative polynomial time (PTIME) distance metric d , computing the explanations is still NP-hard in terms of data complexity for queries that comply to our execution model in Section 3.2. It is sensitive to the choice of admissible parameter changes. For the reparameterizations

shown in Table 6.2, it is intractable. However, we can restrict the allowed reparameterizations in Table 6.2 so that the problem is in PTIME.

Theorem 1

Given a why-not question $\Phi = (Q, D, t)$ and a set of operators from the query Q , testing the membership of the operator set in $\mathcal{E}(\Phi)$ is NP-hard in the size of D for queries that only consist of the operators aggregation, projection, renaming, and join. The problem is in PTIME if aggregation functions are restricted to the default ones in SQL.

Proof 1 (Proof Sketch)

We prove Theorem 1 for queries with aggregation through a reduction from set cover and sketch a brute force algorithm for the PTIME result. The full proof is available in [DLHG21b].

Since we can restrict the allowed reparameterizations in Table 6.2 so that computing explanations in PTIME is possible, we introduce a novel algorithm to compute them in the next section.

6.3 Computing Explanations for missing data

In this section, we introduce the Breadcrumb algorithm that computes explanations for missing data in the query result. Breadcrumb restricts the operator reparameterizations as described in the previous section to achieve PTIME complexity on the data. To be efficient in practice, Breadcrumb further leverages novel heuristics to approximate explanations \mathcal{E}^{\approx} that relax \mathcal{E} . After introducing Breadcrumb's four steps to compute \mathcal{E}^{\approx} in detail, we conclude this section with a discussion (i) on the relation between \mathcal{E}^{\approx} and \mathcal{E} from Definition 6.9, (ii) on Breadcrumb's runtime complexity, and (iii) Breadcrumb's scalability features on big data analytics systems.

Algorithm 10 shows the Breadcrumb algorithm that computes the approximate explanations \mathcal{E}^{\approx} based on the why-not question $\Phi = (Q, D, t)$. It has four main phases. In the first phase (l. 1), Breadcrumb computes a set of tuples \bar{T} over the input relations R of query Q in the why-not question $\Phi = (Q, D, t)$.

Algorithm 10: Breadcrumb(Φ)

Input: WhyNot question $\Phi = (Q, D, t)$

Output: A set of explanations \mathcal{E}^\approx

- 1 $\langle \mathcal{M}_{sbt}, \overline{T} \rangle \leftarrow \text{schemaBacktracing}(\Phi)$
 - 2 $\mathcal{S} \leftarrow \text{schemaAlternatives}(\mathcal{M}_{sbt}, \overline{T}, \Phi)$
 - 3 $R^A \leftarrow \text{dataTracing}(\mathcal{S}, \Phi)$
 - 4 $\mathcal{E}^\approx \leftarrow \text{computeApproximateExplanations}(R^A, \mathcal{S}, \Phi)$
 - 5 **return** \mathcal{E}^\approx
-

These tuples describe the data that potentially contribute to the missing, but expected result. The algorithm further computes a mapping \mathcal{M}_{sbt} which associates each attribute in t and each attribute referenced in the parameters of an operator of Q with a set of attributes from the input. That is necessary to replace attributes in the operators with alternative attributes. We refer to these input attributes as *source attributes*.

In the second phase, Breadcrumb computes alternatives for each source attribute in \mathcal{M}_{sbt} to account for attributes that may not have been chosen appropriately in the query Q (l. 2). Based on these alternatives, it generates a set of *schema alternatives* \mathcal{S} . Each schema alternative corresponds to a possible reparameterization of attribute references in Q .

In the third phase (l. 3), Breadcrumb traces data from the input D through the operators of Q to the query result. It extends each operator in Q to compute annotations that account for the reparameterizations. These annotations describe, e.g., which tuple exists for which schema alternative and which values each tuple holds under each schema alternative. Breadcrumb needs these annotations to compute the explanations in the next phase.

In the final phase, Breadcrumb computes the approximate explanations \mathcal{E}^\approx from the previously collected annotations (l. 3). In the following, we describe each of the four phases in more detail.

6.3.1 Phase 1: Schema backtracing

Breadcrumb starts the computation with the *schemaBacktracing*(Φ) method. It takes the why-not question $\Phi = (Q, D, t)$ as input to compute (i) the set of NIPs \bar{T} (Definition 6.1) that are potentially relevant to produce t under some reparameterizations and (ii) the attribute mapping \mathcal{M}_{sbt} that associates each attribute in Q 's output with the set of attributes from the input and intermediate results. This mapping helps to indentify valid reparameterizations in the query. Breadcrumb's first phase is data-independent.

Breadcrumb utilizes the schema matching algorithm in Section 4.3.1 to match the attributes of the missing tuple t in the why-not question onto the schema of the result relation $\llbracket Q \rrbracket$. For that purpose, we extend the tree-pattern syntax to also support the instance placeholder $?$ and the multiplicity placeholder $*$. Once it obtained the schema match, it transfers all constraints from t to the match to initiate the actual schema backtracing in the next step.

Breadcrumb's schema backtracing resembles Pebble's backtracing algorithm in Section 5.3. The biggest difference between these algorithms is that Breadcrumb's schema backtracing is data-independent. Thus, it applies the same structural manipulations to the schema-match as Pebble, but does not join tuple identifiers or replace identifier placeholders with tuple identifiers. Breadcrumb preserves the constraints from t when possible, e.g., when an attribute is renamed. It further removes the constraint when needed, e.g., when a constraint is defined over an aggregated value. In addition to manipulating t , Breadcrumb updates the mapping \mathcal{M}_{sbt} alongside the backtracing.

Before returning from the schema matching algorithm Breadcrumb has obtained $\bar{T} = \{\bar{t}_{R_1}, \dots, \bar{t}_{R_n}\}$. It contains one NIP \bar{t}_{R_i} for each of Q 's input relations R_1 to R_n . This NIP describes all tuples in the input relation R_i that potentially contribute tuples in the query output $\llbracket Q \rrbracket$ matching the missing tuple t under some schema alternative.

The output \bar{T} is coupled with a mapping \mathcal{M}_{sbt} that associates each attribute $t.A$ of the why-not tuple t with source attributes from the schema of

R_1, \dots, R_n . To identify source attributes potentially relevant for operator reparameterizations, Breadcrumb further adds an entry to \mathcal{M}_{sbt} for influencing attributes $O.A$ referenced by operator O .

To denote the two different association types, we apply color-coding. When we refer to a source attribute $\bar{t}.X$ referencing an attribute in the why-not tuple $t.A$, we denote $\frac{A}{X}$. Associations between a source attribute $\bar{t}.X$ and an attribute $O.A$ are written as $\frac{O.A}{X}$. Further, we represent a pair $(\bar{t}, \mathcal{M}_{sbt})$ as a single nested tuple mirroring the nesting structure of \bar{t} by using mappings from \mathcal{M}_{sbt} rather than attribute names. For instance, if $\frac{A}{X}$, $\frac{O.A}{X}$, and $\frac{O.B}{X}$, then we substitute X with $\frac{A,O.A,O.B}{X}$.

Example 15

We illustrate the schema backtracing on the running example. Recall that the example why-not tuple is:

$$t = \langle city : \text{“NY”}, nList : \{\{*\}\}\rangle$$

Further, the example query $Q_{example}$ computes the result exclusively from the input relation $R_{example}$ in Table 1.1.

Breadcrumb utilizes the schema-matching algorithm from the tree-pattern matching algorithm to match the attributes in t on the result schema, before it starts the schema backtracing. The schema backtracing yields $\mathcal{M}_{sbt_{example}}$ and $\bar{T}_{example} = \{\bar{t}_{example}\}$, which we show in combined fashion:

$$\bar{t}_{example} = \left\langle \frac{t.nList, \pi.name, \mathcal{N}.nList, \mathcal{N}.name}{name} : ?, \frac{F.address1}{address1} : \{\{ \langle \frac{t.city, \pi.city}{city} : \text{“NY”}, \frac{\sigma.year}{year} : ? \rangle \}\}, address2 : ? \right\rangle$$

The denominators describe the attributes of the tuples in the input relation $R_{example}$. The blue numerators indicate attributes in the result tuple that match attributes in the why-not tuple t . In this example the *name* attribute and the nested attribute *address1.city* refer to attributes appearing in t . The latter requires the specific value “NY” to match t . The red numerators indicate attributes accessed by operators in $Q_{example}$. For instance, the red numerator *F.address1* indicates that the flatten operator accesses *address1*. Then, Breadcrumb should consider alternatives for attribute *address1* to find all

reparameterizations. Similarly, *Breadcrumb* should consider alternatives for the nested attribute *year*, even though it does not appear in the why-not tuple. However, the selection accesses the *year*.

6.3.2 Phase 2: Schema alternatives

When *Breadcrumb* returns from schema backtracing, it proceeds with determining schema alternatives. A schema alternative replaces zero or more attributes with alternatives in the query’s operator parameters. The schema alternatives ensure that *Breadcrumb* considers all possible reparameterizations that involve changing an attribute to another. The set of all schema alternatives covers all attribute replacements. *Breadcrumb* leverages the schema alternatives since they encode sets of query reparameterizations. They represent all reparameterizations that replace certain attributes in the query with certain alternative attributes. For instance, the alternatives account for misinterpreted attributes in the query.

Identifying attribute alternatives. To find the schema alternatives, *Breadcrumb* starts with identifying alternatives for individual attributes in the NIPs \bar{T} . For each attribute X in each $\bar{t}_{R_i} \in \bar{T}$, *Breadcrumb* computes a set of alternative attributes \mathcal{X}' :

$$\mathcal{X}' := \{X'_1, \dots, X'_k\}, \text{ where } X'_j \in R_i \wedge \text{type}(X) = \text{type}(X'_j)$$

Recall that we restrict the reparameterizations to operator parameter. We do not allow for changing the query structure. Therefore, *Breadcrumb* only considers attribute alternatives in the same relation because replacing an attribute with an attribute from another relation generally requires more changes to the query than the reparametrizations in Table 6.2. *Breadcrumb* obtains the set of alternatives \mathcal{X}' for each attribute X as input parameters. Attribute alternatives can be determined based on the data type of attributes, by the user asking the why-not question, or using schema matching techniques [ADMR05; DR02]. This approach ensures that *Breadcrumb* only

considers meaningful alternatives, which is necessary to avoid blowing up computations through an impractically high number of alternatives.

Enumerating and pruning schema alternatives. Breadcrumb computes the schema alternatives from the attribute alternatives. Formally defined, a schema alternative is a tuple

$$S := \langle \bar{T}, \mathcal{M} \rangle$$

that holds a set of NIPs \bar{T} and a mapping \mathcal{M} . The set \bar{T} holds one NIP per table accessed by Q . The mapping \mathcal{M} resembles \mathcal{M}_{sbt} since for each attribute $X_i \in \bar{t}_j \wedge \bar{t}_j \in \bar{T}$ it records the operators accessing X_i and the output attributes in Q 's result to which X_i contributes. Schema alternatives replace attributes of one operator independently from attributes of another operator. Thus, some schema alternatives may alter the query's output schema or lead to an invalid query, e.g., containing operators that reference non-existing attributes. Breadcrumb prunes these schema alternatives. It only considers schema alternatives that preserve the output schema and yield valid reparameterizations for all operators in Q .

Example 16

In our running example, Breadcrumb obtains the following attribute alternatives:

$$\begin{aligned} name' &= \{name\} \\ address1' &= \{address2, address1\} \\ city' &= \{address2.city, address1.city\} \\ year' &= \{address2.year, address1.year\} \end{aligned}$$

Based on these attribute alternatives, Breadcrumb incrementally enumerates all schema alternatives as shown in Figure 6.2. It begins with the flatten operator, for which it either picks the original attribute `address1` or its alternative `address2` as the attribute to be flattened. For either version of the reparameterized flatten operator, the selection operator can refer to `address1.year` or

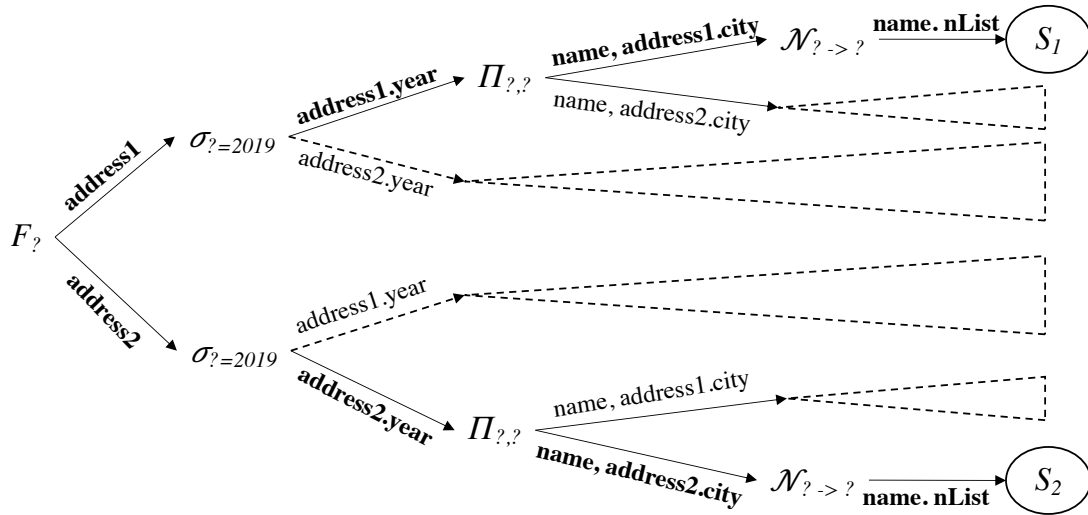


Figure 6.2: Enumerating and pruning schema alternatives (adapted from [DLHG21a])

address1.year, given the alternatives for *year*'. Breadcrumb continues the search until it has assessed all alternatives for all operators.

While the solid lines in Figure 6.2 indicate the derivation of possible schema alternatives, the dashed lines and subtrees in Figure 6.2 describe pruned alternatives. They are pruned, for they address attributes that do not exist at this stage in the example query Q_{example} . When Breadcrumb flattens *address1*, it can only access the alternative *address1.year* for *year* in the selection since *address2.year* is still nested. Breadcrumb further prunes alternatives that alter the output schema because the schema is fixed by definition. Let us assume *address1* has an additional nested attribute *city1*. If Breadcrumb uses an attribute *address1.city1* instead of *address1.city* flattening *address1* changes the output schema to $\{\{\langle \text{city1}, \text{nList} \rangle\}\}$, which alters the query's output schema.

Finally, two schema alternatives remain, denoted as S_1 and S_2 in Figure 6.2:

$$S_1 = \langle \{\bar{t}_1\}, \mathcal{M}_1 \rangle$$

$$S_2 = \langle \{\bar{t}_2\}, \mathcal{M}_2 \rangle$$

In S_1 , \bar{t}_1 is the same as $\bar{t}_{example}$ shown in Example 15 and \bar{t}_2 replaces the address attribute:

$$\bar{t}_2 = \left\langle \frac{t.nList, \pi.name, \mathcal{N}.nList, \mathcal{N}.name}{name} :?, \right. \\ \left. address1 :?, \frac{F.address2}{address2} : \{ \{ \langle \frac{t.city, \pi.city}{city} : "NY", \frac{\sigma.year}{year} :? \rangle \} \} \right\rangle$$

Given the schema alternatives, Breadcrumb is ready to advance to the third phase.

6.3.3 Phase 3: Data tracing

In the third phase, Breadcrumb identifies and traces the data that potentially yields the missing tuple in the why-not question. Breadcrumb instruments the operators in Q to simultaneously execute the query under all previously computed schema alternatives. It ensures that the attributes referenced in any alternative are retained in the output of each operator as long as they are needed. Furthermore, it adds annotations to the processed data so that it can extract the query results for each schema alternative.

Similarly to Pebble (cf. Chapter 5), Breadcrumb extends each operator with an operator-specific tracing procedure. The procedure takes the operator O , an annotated relation R^A , and the schema alternatives \mathcal{S} as input and yields an annotated relation $R^{A'}$ and an updated set of schema alternatives \mathcal{S}' as output. Breadcrumb modifies the default operator semantics to include result tuples that could be produced by possible reparameterizations for each schema alternative. Like Pebble, it extends the operator semantics to track provenance annotations. These annotations are more comprehensive than Pebble's annotations since (i) they apply to each schema alternative individually and (ii) record additional information that Breadcrumb needs to compute explanations in Breadcrumb's last phase.

We distinguish the following types of annotation columns that Breadcrumb adds to each operator output $R^{A'}$. t' denotes a result tuple in $R^{A'}$.

- *id*: Similarly to Pebble, Breadcrumb assigns a unique identifier to each top-level tuple. Breadcrumb uses the identifiers to trace data-dependencies correctly. That is necessary to identify distinct tuples while computing explanations in the last phase.
- *validS_i*: For each schema alternative S_i , the boolean annotation *validS_i* describes whether t' is part of the output of an operator under schema alternative S_i . Since Breadcrumb considers multiple schema alternatives simultaneously, Breadcrumb sets *validS_i* only to `true`, if the top-level output tuple t' in joint output R^A exists under schema alternative S_i . Not every tuple exists under all alternatives simultaneously.
- *consistentS_i*: For each schema alternative S_i , the *consistentS_i* boolean annotation identifies if a tuple t' potentially contributes to the missing answer.
- *retainedS_i* indicates if t' is an output tuple in the query under schema alternative S_i without further operator reparameterizations, such as modifying the filter condition beyond replacing the attributes according to schema alternative S_i . If t' is such a tuple, Breadcrumb sets *retainedS_i* to `true`, or `false`, otherwise.

While the tuple identifier is independent of the schema alternative. The latter three annotations depend on the schema alternative. Let us assume, we have $s = |S|$ schema alternatives. Then, Breadcrumb computes $1 + (3 \cdot s)$ annotations for each top-level tuple in each operator output.

Breadcrumb adds the annotations as columns to the operator's output relation R^A using the *annotate* method shown in Algorithm 11. Given a tuple t , an annotation-value mapping *avMap*, a schema alternative S_i , and an operator O , the *annotate* method returns an annotated tuple t' . It creates one attribute for each entry in the *avMap* with a unique *label* generated from the input parameters and concatenates it with t to create t' .

Given the above preliminaries, we describe Breadcrumb's tracing procedures for the operators used in our running example, omitting the projection operator since it simply propagates the identifiers S_i , the *consistentS_i*

Algorithm 11: $\text{annotate}(t, avMap, S_i, O)$

Input: $t, avMap, S_i, O$ **Output:** t'

```
1 foreach  $\langle a : v \rangle \in avMap$  do
2    $label \leftarrow a + \text{"S"} + i + \_ + \text{id}(O)$ 
3    $t' \leftarrow t \circ \langle label : v \rangle$ 
4 return  $t'$ 
```

Algorithm 12: $\text{tableAccess}(O, R, S)$

Input: O, R, S **Output:** R^A

```
1  $R^A \leftarrow \emptyset$ 
2 foreach  $t \in R$  do
3    $t' \leftarrow t \circ \langle \text{"id"} + \text{id}(O) : \text{id}(t) \rangle$ 
4   foreach  $S_i = \langle \bar{T}_i, \mathcal{M}_i \rangle \in S$  do
5      $c \leftarrow t \simeq \bar{t}_R \in \bar{T}_i$ 
6      $t' \leftarrow \text{annotate}(t', \{\{\langle \text{consistent} : c \rangle\}\}, S_i, O)$ 
7    $R^A \leftarrow R^A \cup \{\{t'\}\}$ 
8 return  $\langle R^A, S \rangle$ 
```

columns, and $validS_i$ columns from its input to its output. Our implementation features tracing algorithms for all operators from Table 3.1.

Table access. The tracing procedure of the table access operator adds the necessary annotations to all tuples in the input relation. Algorithm 12 shows the annotation algorithm. The input relation R is the original input relation, which has no annotations so far. Further, the algorithm obtains the operator O and the set of schema alternatives S for annotation purposes.

The algorithm iterates over each top-level tuple of the input relation R , which is gradually extended with additional annotation attributes. First, it adds an id attribute that stores a unique tuple identifier. Then, the algorithm attaches a $consistentS_i$ attribute for each schema alternative S_i , using the

id_0	firstname	lastname	address1			address2			c_S1_0	c_S2_0
2	Sue	Miller	city		year	city		year	1	1
			31	LA	2019	41	NY	2019		
			32	NY	2018	42	LA	2018		
3	Sue	Walker	city		year	city		year	0	1
			51	SF	2018	61	LV	2017		
			52	LA	2019	62	NY	2019		

Table 6.7: Shortened input data R_{short} from Table 1.1 after annotation

annotate method in Algorithm 11. The attribute’s value c indicates whether a tuple potentially contributes to the missing data. It is `true` when t matches the tuple \bar{t}_R in the set of tuples \bar{T}_i of schema alternative S_i , otherwise, it is `false`. To assess whether t matches \bar{t}_R , the algorithm utilizes the data matching algorithm introduced in Section 4.3.2. Once the algorithm has computed all annotations for all top-level tuples $t' \in R^A$, it returns R^A together with the S . The latter is unmodified since the table access operator does not change the structure of its input.

Example 17

We apply the tracing procedure on the running, but strip the example input. Like in Chapter 5, we reduce the input data in Table 1.1 to the two “Sue” tuples in Table 6.7. The shown table has the known top-level attributes *firstname*, *lastname*, *address1*, and *address2*. In addition to them, it has three further columns: (i) *id_0*, (ii) *c_S1_0*, and (iii) *c_S2_0*. The postfix 0 in the column names indicate the operator identifier 0. S1 and S2 in the latter two columns describe the schema alternatives S_1 and S_2 . The c is an abbreviation for *consistent*. Thus, column (i) holds the unique identifier, and columns (ii) and (iii) indicate whether the tuple is consistent under schema alternative S_1 or S_2 , respectively. For conciseness, the value 1 indicates *true* and the value 0 represents *false* in these columns.

In the example, tuple 2 is consistent under both schema alternatives S_1 and S_2 since it holds nested *city* “NY” in *address1* and another one in *address2*. Tuple 3 only holds the *city* “NY” in *address2*. Thus, it is only consistent under schema alternative S_2 .

Algorithm 13: $\text{flatten}(O, R^A, \mathcal{S})$

Input: O, R^A, \mathcal{S} **Output:** $\langle R^A, \mathcal{S}' \rangle$

- 1 For each $S_i \in \mathcal{S}$, let U_i be the result of executing O wrt S_i and generalized to an outer flatten
 - 2 For each $S_i \in \mathcal{S}$, let $S'_i = \langle \bar{T}'_i, \mathcal{M}'_i \rangle$ be the schema alternative reflecting the flattening wrt S_i
 - 3 $\mathcal{S}' \leftarrow \bigcup_{S_i \in \mathcal{S}} S'_i$
 - 4 $R^A \leftarrow \emptyset$
 - 5 **foreach** $S'_i \in \mathcal{S}'$ **do**
 - 6 **foreach** $t \in U_i$ **do**
 - 7 $v \leftarrow$ value of most recent $\text{valid}_{S'_i}$ attribute
 - 8 $r \leftarrow t$ is in the result of original flatten wrt S_1
 - 9 $c \leftarrow t \simeq \bar{t}_U$, where $\bar{t}_U \in \bar{T}'_i$
 - 10 $\text{avMap} \leftarrow \{ \langle \text{valid} : v \rangle, \langle \text{retained} : r \rangle, \langle \text{consistent} : c \rangle \}$
 - 11 $U'_i \leftarrow U'_i \cup \{ \text{annotate}(t, \text{avMap}, S'_i, O) \}$
 - 12 $\bar{t} \leftarrow \bar{t}_U \in \bar{T}'_i$
 - 13 $R^A \leftarrow \text{merge}(R^A, U'_i, S_i, O, \bar{t})$
 - 14 **return** $\langle R^A, \mathcal{S}' \rangle$
-

Flatten. Since the flatten operator is the first operator in the example query pipeline in Figure 6.1, we discuss the flatten operator next. Algorithm 13 describes the corresponding tracing procedure. The tracing procedure considers the results U_i of the flatten operator under all schema alternatives $S_i \in \mathcal{S}$. It applies an outer flatten to obtain U_i since changing an inner flatten to an outer flatten is a possible reparameterization. Next, it updates the schema alternatives S_i to S'_i to reflect the restructuring of tuples through flattening. After computing \mathcal{S}' and initializing the output relation R^A to an empty relation, the algorithm iterates over all the schema alternatives S'_i (ll. 5-13) to assign the proper annotations to each top-level tuple t in the flattened U_i , which belongs to the schema alternative S'_i (ll. 6-11).

It determines the values v , c , and r for the *valid*, *consistent*, and *retained* annotations by evaluating boolean conditions. All values are `false` by default. The algorithm further preserves the *valid* value from the input data. If an input tuple had $valid = \text{true}$, the corresponding output tuple also has $valid = \text{true}$. Further, $r = \text{true}$ if t is in the result of the original, e.g., inner flatten with respect to S'_i . The value $c = \text{true}$ if t matches the NIP \bar{t}_U that is associated with the schema alternative S'_i . The algorithm adds *valid*, *consistent*, and *retained* annotations to the *avMap* and calls the *annotate* function (Algorithm 11) to add the annotations to t . Then, it adds the annotated tuple to the updated flatten result U'_i .

In the final step of the loop over all schema alternatives S'_i (l. 13), the algorithm calls the *merge* function. It merges the current annotated flatten result U'_i with the flatten results from the other schema alternatives into R^A . The *merge* implements a concatenation of tuples with the same *id* annotation across the outer flatten results of all schema alternatives. It ensures not to replicate columns that remain the same across all schema alternatives. Since the number of tuples with a given *id* may vary after the flatten due to varying cardinalities of the alternative nested relations, the *merge* method pads missing concatenation partners with null values (\perp). Finally, it assigns each top-level output tuple a new unique *id* for further identification and returns the merged relation R^A together with the updated schema alternatives S' .

id_1	firstname	lastname	address1			address2			city	year	city_S2	year_S2	...	c_S1_1	r_S1_1	v_S1_1	c_S2_1	r_S2_1	v_S2_1
<i>113</i>	Sue	Miller	<i>31</i>	LA	2019	<i>41</i>	NY	2019	LA	2019	NY	2019	...	0	1	1	1	1	1
			<i>32</i>	NY	2018	<i>42</i>	LA	2018											
<i>114</i>	Sue	Miller	<i>31</i>	LA	2019	<i>41</i>	NY	2019	NY	2018	LA	2019	...	1	1	1	0	1	1
			<i>32</i>	NY	2018	<i>42</i>	LA	2018											
<i>115</i>	Sue	Walker	<i>51</i>	SF	2018	<i>61</i>	LV	2017	SF	2018	LV	2017	...	0	1	1	0	1	1
			<i>52</i>	LA	2019	<i>62</i>	NY	2019											
<i>116</i>	Sue	Walker	<i>51</i>	SF	2018	<i>61</i>	LV	2017	LA	2019	NY	2019	...	0	1	1	1	1	1
			<i>52</i>	LA	2019	<i>62</i>	NY	2019											

Table 6.8: Output R_F of the tracing procedure for the flatten operator in the example pipeline in Figure 1.1, when applied on R_{short} ; italic numbers indicate tuple identifiers

Example 18

In our running example, the inner flatten takes the relation shown in Table 6.7 as input, together with the schema alternatives from Example 16. It produces the annotated relation summarized in Table 6.8. First, observe that it combines both schema alternatives since both attributes *address1* and *address2* have been flattened. The attributes *city* and *year* originate from *address1*. Unlike *city_S2* and *year_S2* which originate from *address2*, they are referenced in the unmodified example query Q_{example} . Hence, they do not have any suffixes which indicate the association to a schema alternative. Nonetheless, the alternative S_1 , which represents the unmodified schema alternative references these attributes. The alternative S_2 references the attributes *city_S2* and *year_S2* instead of *city* and *year*, respectively. The column marked with ... summarizes all annotation columns of the input, which are treated as “regular” input columns when executing outer flatten for each schema alternative. Let us focus on the new annotations, now.

Under schema alternative S_1 , only tuple 114 has the value 1 in column c_S1_1 . It is the only tuple that is consistent with the why-not question under schema alternative S_1 . Indeed, once S_1 has been processed to reflect the flattening, \bar{T}_1 becomes $\bar{T}'_1 = \{\langle \frac{nList}{name} :?, \frac{city}{city} : \text{“NY”}, year1 :? \rangle\}$. Note that only tuple 114 in Table 6.8 features “NY” in the *city* column and satisfies \bar{T}'_1 . According to the values in columns v_S1_1 and r_S1_1 , the inner flatten (as originally intended) on *address1* yields 4 tuples since they are 1 for all four displayed tuples. If the r_S1_1 annotation was 0 the corresponding tuple would have been lost due to the flatten type (inner flatten rather than outer flatten). If the v_S1_1 annotation was 0 the corresponding tuple could not have been produced under schema alternative S_1 .

Under schema alternative S_2 , the tuples 113 and 116 are consistent (i.e., $c_S2_1 = 1$) with the updated \bar{T}'_2 because both have the value “NY” in the attribute *city_S2*. Furthermore, all tuples are valid and retained as indicated by the 1 values in the v_S1_1 and r_S1_1 columns.

Selection. As shown in Algorithm 14, the tracing procedure for the selection operator adds additional annotation columns to the input relations before

Algorithm 14: $Selection(\sigma_\theta, R^A, \mathcal{S})$

Input: $\sigma_\theta, R^A, \mathcal{S}$
Output: $\langle R^{A'}, \mathcal{S} \rangle$

- 1 $R^{A'} \leftarrow R^A$
- 2 **foreach** $S_i \in \mathcal{S}$ **do**
- 3 $U \leftarrow \emptyset$
- 4 **foreach** $t \in R^{A'}$ **do**
- 5 $v \leftarrow$ value of most recent $valid_{S_i}$ attribute
- 6 $r \leftarrow \theta_{S_i}(t) = \text{true}$
- 7 $c \leftarrow t \simeq \bar{t}_U$, where $\bar{t}_U \in \bar{T}'_i$
- 8 $avMap \leftarrow \{\{\langle valid : v \rangle, \langle retained : r \rangle, \langle consistent : c \rangle\}\}$
- 9 $U \leftarrow U \cup \{\{annotate(t, avMap, S_i, \sigma)\}\}$
- 10 $R^{A'} \leftarrow U$
- 11 **return** $\langle R^{A'}, \mathcal{S} \rangle$

returning it. To take all schema alternatives into account, the algorithm iterates over each schema alternative $S_i \in \mathcal{S}$ (ll. 2-10). It initializes the intermediate output U to an empty relation before iterating through the tuples in the annotated relation $R^{A'}$ (ll. 5-9). In the inner loop, the algorithm extends each tuple with its individual annotations $valid$ (v), $retained$ (r), and $consistent$ (c). It propagates the valid value from the previous $valid$ annotation and assesses the selection condition θ_{S_i} for the $retained$ annotation. That annotation indicates if the tuple survived the selection under the current schema alternative S_i . Note that the algorithm does not remove any tuples, it just marks them with the $retained$ attribute. To obtain the value for the $consistent$ annotation, the algorithm checks whether the current tuple t matches the according NIP \bar{t}_U in S_i . If it is the case, the $consistent$ value becomes 1, otherwise, it is 0. Once the annotations are computed, the algorithm extends t with them and adds t to the intermediate result U . After iterating through all ts in $R^{A'}$, the algorithm assigns U to $R^{A'}$, and repeats the annotation process with the next schema alternative. After all schema alternatives are processed, Algorithm 14 returns the updated $R^{A'}$ and the set

id_2	firstname	...	city	year	city_S2	year_S2	...	c_S1_2	r_S1_2	v_S1_2	c_S2_2	r_S2_2	v_S2_2
113	Sue	...	LA	2019	NY	2019	...	0	1	1	1	1	1
114	Sue	...	NY	2018	LA	2019	...	1	0	1	0	1	1
115	Sue	...	SF	2018	LV	2017	...	0	0	1	0	0	1
116	Sue	...	LA	2019	NY	2019	...	0	1	1	1	1	1

Table 6.9: Output R_σ of the tracing procedure for the selection operator, when applied on R_F in Table 6.8; some attributes are hidden for conciseness

of schema alternatives S . The algorithm leaves the id column unmodified because each input tuple yields exactly one output tuple. Therefore, it does not appear in Algorithm 14.

Example 19

When we apply the tracing procedure in Algorithm 14 on the selection with identifier 2 in the query pipeline in Figure 6.1 on the annotated output R_F in Table 6.8, we obtain the output R_σ in Table 6.9. The procedure creates annotations for both schema alternatives S_1 and S_2 . The attribute values for the consistent attributes c_S1_2 and c_S2_2 as well as the valid attributes v_S1_2 and v_S2_2 are the same as the values in Table 6.8. The values for the retained attributes differ from the values in Table 6.8, because they reflect the selection result. For schema alternative S_1 , the attribute r_S1_2 holds a one for the tuples 113 and 116, because their year is 2019. Similarly, for schema alternative S_2 , the attribute r_S2_2 holds a one for the tuples 113, 114, and 116, because their year_S2 is 2019. Note that the procedure retains all tuples. It does not remove the tuples that the selection would have removed, i.e. tuples 114 and 115 for S_1 and tuple 115 for S_2 . Instead, their retained value is zero.

Projection, renaming, tuple flatten, and tuple nesting. The successor of the selection in the example pipeline from Figure 1.1 is the projection. Therefore, we describe its tracing procedure next. Since the tracing procedure for the projection, renaming, tuple flatten, and tuple nesting operator are quite similar, we describe common internals once for all of them and highlight operator-specific differences. The tracing procedures of these operators resemble the procedure for the selection shown in Algorithm 14.

There are essentially two big differences between that procedure and the procedures for the listed operators. First, the listed operators manipulate the data structure. Therefore, the procedures must update the schema alternatives right at the beginning of the procedure. Each operator conducts different structural manipulations. Thus, each operator's procedure has its individual update routine. Nonetheless, the update routine is essentially the same routine as the one in the relation flatten procedure in Algorithm 13 for all operators. Second, the mentioned operators yield exactly one output tuple for each input tuple. Hence, the procedure sets the *retained* annotation to 1 for all tuples. The rest of the procedure in Algorithm 14 remains unchanged for all listed operators.

The *valid* and *consistent* annotations are computed exactly in the same way as shown in Algorithm 14. Furthermore, the *id* annotation remains untouched for the same reasons as in Algorithm 14. Due to the significant similarity to the selection operator's procedure, we skip a detailed example and move on to the relation nesting procedure.

Relation nesting. The relation nesting operator is the last operator in our running example. Recall that it nests the values of a specified attribute into a nested relation for all tuples in the input relation that hold the same values in all other attributes.

The tracing procedure for the relation nesting is shown in Algorithm 15. Given the operator $\mathcal{N}_{B \rightarrow C}^R$, the annotated input relation R^A , and the schema alternatives \mathcal{S}' , it computes an annotated output relation $R^{A'}$ and an updated set of schema alternatives \mathcal{S} . The algorithm consists of three major steps. First, it processes the schema alternatives (ll. 1-3). Next, the algorithm computes the intermediate results for each schema alternative independently (ll. 4-13). Eventually, it merges the intermediate results (ll. 14-17).

During the first step, Algorithm 15 identifies the annotation attributes X_i in R^A for each schema alternative S_i . Furthermore, it updates the schema alternatives S'_i to reflect the structural manipulations of the tuples in the input relation. It stores the set of updated schema alternatives in \mathcal{S}' . The algorithm leverages it for the annotation computation.

Algorithm 15: $\text{relationNesting}(\mathcal{N}_{B \rightarrow C}^R, R^A, \mathcal{S})$

Input: $\mathcal{N}_{B \rightarrow C}^R, R^A, \mathcal{S}$

Output: $\langle R^A, \mathcal{S}' \rangle$

- 1 For each $S_i \in \mathcal{S}$, let X_i be the set of annotation columns in R^A belonging to S_i , including the identity column
 - 2 For each $S_i \in \mathcal{S}$, let $S'_i = \langle \bar{T}'_i, \mathcal{M}'_i \rangle$ be the schema alternative reflecting the nesting wrt S_i
 - 3 $\mathcal{S}' \leftarrow \bigcup_{S_i \in \mathcal{S}} S'_i$
 - 4 **foreach** $S'_i \in \mathcal{S}'$ **do**
 - 5 $U_i \leftarrow \llbracket \mathcal{N}_{\langle B_1, PX'_i \rangle \rightarrow \langle C_i, PX'_i \rangle}^R (\rho_{S_i \leftarrow S_1, PX_i \leftarrow PX'_i} (\pi_{S_i \cup PX_i} (\mathcal{N}_{X_i \rightarrow PX_i}^T (R^A)))) \rrbracket$
 - 6 $U'_i \leftarrow \emptyset$
 - 7 **foreach** $t \in U_i$ **do**
 - 8 $c \leftarrow t \simeq \bar{t}_U$, where $\bar{t}_U \in \bar{T}'_i$
 - 9 $v \leftarrow t.PX'_i$ most recent *valid* column is 1 at least once
 - 10 $avMap \leftarrow \{\{ \langle \text{valid} : v \rangle, \langle \text{retained} : 1 \rangle, \langle \text{consistent}, c \rangle \}$
 - 11 $t' \leftarrow \text{annotate}(t, avMap, S'_i, \mathcal{N}_{B \rightarrow C}^R)$
 - 12 $t' \leftarrow \text{nullPad}(t', S'_i, C, \mathcal{S}')$
 - 13 $U'_i \leftarrow U'_i \cup \{t'\}$
 - 14 $U \leftarrow \bigcup_{S'_i \in \mathcal{S}'} U'_i$
 - 15 $\mathcal{C} \leftarrow \bigcup_{S'_i \in \mathcal{S}'} C_i$
 - 16 $\mathcal{A} \leftarrow \bigcup_{S'_i \in \mathcal{S}'} \text{annotation columns in } U'_i$
 - 17 $U' \leftarrow \llbracket \gamma_{\text{firstNonNullValue}(\langle \mathcal{C}, \mathcal{A} \rangle) \rightarrow \langle \mathcal{C}, \mathcal{A} \rangle} (\mathcal{N}_{\langle \mathcal{C}, \mathcal{A} \rangle \rightarrow \langle \mathcal{C}, \mathcal{A} \rangle}^R (U)) \rrbracket$
 - 18 $R^A \leftarrow \emptyset$
 - 19 **foreach** $t \in U'$ **do**
 - 20 $t' \leftarrow t \circ \langle \text{"id"} + \text{id}(O) : \text{id}(t) \rangle$
 - 21 $R^A \leftarrow R^A \cup \{t'\}$
 - 22 **return** $\langle R^A, \mathcal{S}' \rangle$
-

In the second step (ll. 4-13), the algorithm computes the *valid*, *consistent*, and *retained* annotations for each schema alternative as the outer loop indicates. It prepares an intermediate result U_i for each schema alternative (l. 5). For that purpose, it nests all attributes X_i into a dedicated provenance tuple PX_i and renames all attributes occurring in S_i to the attribute names in the original schema alternative S_1 . In the final preparation step for U_i , it computes a nested relation of tuples holding PX_i and the attribute to be

nested B_1 . The algorithm further initializes the intermediate annotated result U'_i to an empty relation. Then, the procedure annotates each tuple $t \in U_i$ with provenance annotations in the inner loop (ll. 7-13). It computes the *consistent* attribute c in the same way the other algorithms do. It matches the tuple t against the updated NIP \bar{t}_U . In case of a match, c becomes 1. Otherwise, it is 0. To compute the *valid* annotation v , Algorithm 15 needs to access the newly created nested relation PX_i . It checks whether any of the most recent *valid* annotations nested in PX_i is 1. The most recent *valid* annotations are those *valid* annotations that the relation nesting's predecessor has assigned to each input tuple of the relation nesting operator. If at least one *valid* annotation with the value 1 exists, it sets v to 1 and to 0, otherwise. The idea behind this annotation approach is that the output tuple can be generated from the input if Breadcrumb can generate at least one valid input tuple. Since the relation nesting operation is not selective, the algorithm sets all *retained* annotation values to 1 for all tuples t . It obtains the tuple t' from annotating t and calls the *nullPad* function since it has to align t' 's schema across all alternatives. This function adds annotation attributes and the attributes C_j holding the nested relations for all but the current schema alternative S_i and fills the added attributes with null values. Additionally, it re-arranges the attributes, such that they have the same order for all schema alternatives. As a final action of the second step, the algorithm adds t' to the intermediate result U'_i , which holds the annotated relation nesting result under schema alternative S_i .

Once Algorithm 15 has computed the intermediate nesting results for all schema alternatives, it merges the intermediate results U'_i (ll. 14-17). For that purpose, it first unites all intermediate results U'_i into the relation U . Now, the procedure could return and computation and Breadcrumb could continue with the next operator. However, the result is blown-up, since the tuples that only distinguish themselves from each other by their annotation and the newly created nested relation potentially occur once in U for each schema alternative. To compute side-effects effectively and keep further computations efficient, the algorithm merges tuples that only distinguish in the mentioned attributes. For that purpose, it collects all attributes with

the newly nested relations into \mathcal{C} and the annotation attributes added in the previous step into \mathcal{A} for a final nesting and aggregation operation. In this operation, the algorithm nests the tuples on all attributes but the attributes in \mathcal{C} and \mathcal{A} (l. 17). For the attributes in \mathcal{C} and \mathcal{A} the aggregation picks the first non-null value by applying the *firstNonNullValue* function. That yields the correctly merged output relation U' for three reasons. First, each value combination for the attributes $\mathcal{M} = (\text{LBL}(U) - \mathcal{A}) - \mathcal{C}$ occurs at most once in the input U , because the nesting in line 5 ensures that. Second, the attributes in \mathcal{A} and \mathcal{C} are distinct for each schema alternative. Third, the attributes that do not belong to a schema alternative are null-padded in the second step of this algorithm. In a final step (ll. 18-21), the algorithm assigns each tuple in U' a new unique identifier id and adds it to the final result R^A , before it returns R^A together with S' .

id_4	city	nList		...	p_S1_4	c_S1_4	r_S1_4	v_S1_4	p_S2_4	c_S2_4	r_S2_4	v_S2_4
91	LA	firstname		0	1	1	⊥	0	0	0
		711	Sue									
92	NY	firstname		1	1	1	⊥	0	0	0
		713	Sue									
93	SF	firstname		0	1	1	⊥	0	0	0
		714	Sue									
94	NY	firstname		...	⊥	0	0	0	...	1	1	1
		715	Sue									
95	LA	firstname		...	⊥	0	0	0	...	0	1	1
		717	Sue									
96	LV	firstname		...	⊥	0	0	0	...	0	1	1
		718	Sue									

Table 6.10: Intermediate relation nesting results after nesting individual schema alternatives; the upper half belongs to alternative S'_1 and the lower half to alternative S'_2

...	c_S1_1	r_S1_1	v_S1_1	c_S1_2	r_S1_2	v_S1_2	...
711	...	0	1	1	0	1	1
712	...	0	1	1	0	1	1

Table 6.11: Detailed captured annotations stored in attribute p_S1_4 of tuple 91 in Table 6.10

city	nList	nList_S2	...	p_S1_4	c_S1_4	r_S1_4	v_S1_4	p_S2_4	c_S2_4	r_S2_4	v_S2_4						
991	LA	<table border="1"> <tr><th>firstname</th></tr> <tr><td>711 Sue</td></tr> <tr><td>712 Sue</td></tr> </table>	firstname	711 Sue	712 Sue	<table border="1"> <tr><th>firstname</th></tr> <tr><td>717 Sue</td></tr> </table>	firstname	717 Sue	0	1	1	⊥	0	1	1
		firstname															
711 Sue																	
712 Sue																	
firstname																	
717 Sue																	
992	NY	<table border="1"> <tr><th>firstname</th></tr> <tr><td>713 Sue</td></tr> </table>	firstname	713 Sue	<table border="1"> <tr><th>firstname</th></tr> <tr><td>715 Sue</td></tr> <tr><td>716 Sue</td></tr> </table>	firstname	715 Sue	716 Sue	1	1	1	⊥	1	1	1
		firstname															
713 Sue																	
firstname																	
715 Sue																	
716 Sue																	
993	SF	<table border="1"> <tr><th>firstname</th></tr> <tr><td>714 Sue</td></tr> </table>	firstname	714 Sue	⊥	0	1	1	⊥	0	0	0			
firstname																	
714 Sue																	
994	LV	⊥	<table border="1"> <tr><th>firstname</th></tr> <tr><td>718 Sue</td></tr> </table>	firstname	718 Sue	...	⊥	0	0	0	...	0	1	1			
firstname																	
718 Sue																	

Table 6.12: Relation nesting result $R_{\mathcal{N}^R}$ after merging the individual results in Table 6.10

Example 20

We run the tracing procedure for the relation nesting operator on our reduced running example. The procedure first updates the alternatives S_1 and S_2 to S'_1 and S'_2 to incorporate that the *firstname* attribute is nested into the newly created *nList* attribute. Next, it iterates through both schema alternatives. For each of them, it computes independent intermediate results as shown in Table 6.10. The upper half belongs to the first schema alternative S'_1 and the lower half to S'_2 . Note that the non-annotation-related attributes all share the names of the attributes referenced in the original schema alternative S'_1 . This is necessary to merge the tuples later. The procedure has also added annotations to the top-level tuples according to Algorithm 15. We describe the annotations one by one starting with the valid attributes v_S1_4 and v_S2_4 . In the upper half v_S1_4 is 1 for all tuples, in the lower half it is 0 for all tuples because the tuples in the lower half are computed from the S'_2 . Thus, they cannot be derived from S'_1 . For the same reason, values of v_S2_4 are 0 in the upper half and 1 in the lower half. Since the tuple nesting operator does not remove any items from the result the retained attributes r_S1_4 and r_S2_4 are 1 for all valid tuples under the associated schema alternative. Furthermore, the tracing procedure has re-evaluated the valid tuples regarding their compatibility to the why-not tuple. Since the why-not tuple requires the *city* attribute to hold the value NY, only the tuples 92 and 94 hold the value 1 in the consistent attributes c_S1_4 and c_S2_4 , respectively.

The intermediate result further holds nested relations for the provenance annotations, called p_S1_4 and p_S2_4 . Table 6.11 shows the nested annotations in p_S1_4 in tuple 91. As the identifiers 711 and 712 show, they are associated with the two nested *firstnames* Sue in the same top-level tuple. The consistent, retained, and valid attributes are the ones introduced by the previous operators. They are displayed in Table 6.8 and Table 6.9.

Once all intermediate results are computed, the procedure merges the tuples in Table 6.10 into the tuples shown in Table 6.12. It creates a dedicated attribute for each newly created nested relation. In our example, these are $nList$ for S'_1 and $nList_S2$ for S'_2 . However, it keeps the non-modified attributes, such as *city* in our example, a shared attribute across schema alternatives. In fact, the tracing procedure even merges the attributes based on these attributes. When tuples share the same values in the non-modified attributes. That is why tuple 991 in Table 6.12 holds the $nLists$ of tuples 91 and 95 in Table 6.10.

The merge process further integrates the annotations. It retains the values of those tuples whose valid attributes are set to 1. In tuple 991 the procedure has transferred the annotation attributes p_S1_4 , c_S1_4 , r_S1_4 , v_S1_4 from tuple 91 and p_S2_4 , c_S2_4 , r_S2_4 , v_S2_4 from tuple 95. If the values of the non-modified attributes only occur in a tuple of a single schema alternative, such as the *city*LV in tuple 994, the merge process fills the provenance attributes of the other alternatives with zeros or null values.

Eventually, the procedure returns the nested relation R_{NR} in Table 6.12 together with the updated set of schema alternatives S' , which holds S'_1, S'_2 .

Even though we have described the majority of supported operators while walking through the running example, the procedures for the aggregation, join, and union operators are not addressed, yet. In the following, we briefly sketch their internals.

Aggregation. The tracing procedure for the aggregation operator is similar to the procedure for the selection operator described in Algorithm 14 because the aggregation also yields one output tuple per input tuple. Like the selection procedure applies the filter condition on all alternatives, the aggregation procedure applies the aggregation on all schema alternatives simultaneously.

In addition to the selection procedure, the aggregation procedure updates the structure of the schema alternatives. The rest of the procedure is analogous to the procedure in Algorithm 14.

Join and Union. Unlike the operators described so far, the join and union operators have two input relations. Therefore, both tracing procedures need to combine not only two input relations but also two sets of schema alternatives. Both operators compute the cross-product of the two input schema alternatives to obtain the output schema alternatives. Note that the cross-product only applies to the alternatives, not the processed data.

Based on these newly created alternatives the join procedure combines the tuples of the two input relations as follows. It rewrites the join to an outer join to identify tuples marked *consistent* that get removed since the join is too rigid. Further, it rewrites the join condition for each newly created alternative and combines them via disjunctive concatenation. Afterward, it applies the same tuple checks as the selection procedure in Algorithm 14. To identify the value for the *valid* attribute, it combines the values of the two most recent *valid* attributes from each input tuple. Only if both are 1, the new *valid* attribute becomes 1. Otherwise, it is 0.

The procedure for the union operator first aligns the schemata of both input relations. While the input relations have matching schemata before Breadcrumb annotates them, Breadcrumb adds attributes to account for the schema alternatives and the provenance annotations. Thus, it applies the same null-padding as in Algorithm 15 to align the schemata. Then, it applies the union operator and annotates the resulting tuples as in Algorithm 14. However, this procedure sets the *valid* attribute to 1, if either of the two *valid* attributes is 1. Otherwise, it is 0.

Once Breadcrumb has executed the tracing procedure for all operators in the query, it holds a nested relation, which extends the original query result in two aspects. First, the relation may hold additional tuples and attributes that could belong to the result under some schema alternatives. Second, it holds the annotations that Breadcrumb leverages in the next step to compute approximated explanations.

Algorithm 16: *computeApproximateExplanations*(R^A, \mathcal{S}, Φ)

Input: $\langle R^A, \mathcal{S}, \Phi \rangle$ **Output:** \mathcal{E}^\approx

```
1  $\mathcal{E}^\approx = \emptyset$ 
2 foreach  $i \in \{1, \dots, |\mathcal{S}|\}$  do
3    $S_i \leftarrow$   $i$ th schema alternative in  $\mathcal{S}$ 
4    $E_{S_i} \leftarrow$  set of operators modified by  $S_i$ 
5    $R_{S_i}^A \leftarrow$  recursively flatten nested provenance relations associated with  $S_i$ 
6    $OID \leftarrow$  set of operator identifiers in  $\Phi.Q$ 
7    $\theta_{S_i} \leftarrow$  false
8    $RA \leftarrow \emptyset$ 
9   foreach  $j \in OID$  do
10     $\theta_{S_i} \leftarrow \theta_{S_i} \vee ((v_{S_i_j} = 1 \wedge c_{S_i_j} = 1) \vee (v_{S_i_j} = \perp \wedge c_{S_i_j} = \perp))$ 
11     $RA \leftarrow RA \cup r_{S_i_j}$ 
12     $R_{S_i}^A \leftarrow \llbracket \delta(\pi_{RA}(\sigma_{\theta_{S_i}}(R_{S_i}^A))) \rrbracket$ 
13     $\mathcal{E}_{S_i}^\approx \leftarrow$  for each tuple  $t$  in  $R_{S_i}^A$ , extract all operator identifiers  $j$  for which
       $r_{S_i_j} = 0$ 
14     $\mathcal{E}_{S_i}^{\approx'}$   $\leftarrow \emptyset$ 
15    foreach  $E \in \mathcal{E}_{S_i}^\approx$  do
16       $\mathcal{E}_{S_i}^{\approx'} \leftarrow \mathcal{E}_{S_i}^{\approx'} \cup \{E_{S_i} \cup E\}$ 
17     $\mathcal{E}^\approx \leftarrow \mathcal{E}^\approx \cup \mathcal{E}_{S_i}^{\approx'}$ 
18 Prune  $\mathcal{E}^\approx$  based on upper and lower bound of side-effects for each explanation
    in  $\mathcal{E}^\approx$  and sort them according to the partial order defined by Definition 6.7.
19 return  $\mathcal{E}^\approx$ 
```

6.3.4 Phase 4: Computing approximate explanations

In the final step, Breadcrumb leverages Algorithm 16 to approximate the set of explanations formally defined in Definition 6.9. The algorithm takes the annotated relation R^A , the set of schema alternatives \mathcal{S} , and the why-not question Φ as input. It computes and returns the approximate explanations \mathcal{E}^\approx , which holds sets of operators. If the operators in the sets are reparameter-

ized in the right way, the missing answer appears in the result. Breadcrumb does not provide the reparameterization though.

After initializing \mathcal{E}^\approx (l. 1), Breadcrumb computes the approximate explanations for each schema alternative S_i individually (ll. 2-17) and merges them into \mathcal{E}^\approx . In a final step, Algorithm 16 prunes \mathcal{E}^\approx based on upper and lower bounds of side-effects for each explanation E in \mathcal{E}^\approx . It sorts the explanations according to the partial order in Definition 6.7 (l. 18).

Algorithm 16 iterates through the schema alternatives' indices (l. 2) when computing the explanations for each schema alternative, because the annotations have the index encoded in their attribute name. It obtains the schema alternative S_i at index i for further processing and collects a set of operators E_{S_i} that are reparameterized in schema alternative S_i . This set holds all operators whose attribute references have been replaced with an alternative attribute by S_i . Then, the algorithm conducts preparations needed to obtain the explanations based on S_i . It recursively flattens the nested provenance collections associated with S_i into $R_{S_i}^A$ (l. 5), obtains the set of operator identifiers OID in the query $\Phi.Q$, initializes a selection condition θ_{S_i} to false, and creates an empty attribute set RA . After the preparation, Breadcrumb iterates through the operator identifiers j to extend the selection condition θ_{S_i} and the attribute set RA . The selection condition θ_{S_i} ensures that Breadcrumb only considers those tuples in $R_{S_i}^A$ that are *valid* and *consistent*. Both attributes have to hold either 1 or the null value \perp . The 1-values represent *valid* and *consistent* result tuples that match the why-not question. However, Breadcrumb also has to check for null-values since operators with multiple inputs may generate null-values. For instance, the join generates them for tuples without a join partner. Furthermore, Breadcrumb adds the operator- and alternative-dependent *retained* annotations to RA . After the loop over the operators in OID , Breadcrumb applies θ_{S_i} in a selection and RA in a projection onto $R_{S_i}^A$. It further deduplicates the remaining tuples and updates $R_{S_i}^A$ (l. 12). $R_{S_i}^A$ now only holds *retained* annotations. Therefore, each tuple in $R_{S_i}^A$ represents an explanation for the missing data in the result. Breadcrumb iterates through the tuples in $R_{S_i}^A$. For each tuple, it extracts the operator identifiers j for which the *retained* attribute holds a zero (l.

13) and stores the obtained set of operator identifiers in $\mathcal{E}_{S_i}^{\approx}$. Since Breadcrumb has to account for the operators in E_{S_i} that are reparameterized by the schema alternative S_i it loops through the explanations $\mathcal{E}_{S_i}^{\approx}$ and merges the operators E_{S_i} with the operators in the explanations in $\mathcal{E}_{S_i}^{\approx}$ (ll. 14-16). Now, the resulting $\mathcal{E}_{S_i}^{\approx'}$ holds the set of approximate explanations for schema alternative S_i . Breadcrumb adds the set to \mathcal{E}^{\approx} (l. 17) before advancing to the next schema alternative.

Once Breadcrumb has iterated through alternatives, it prunes and sorts the explanations in \mathcal{E}^{\approx} , before it returns \mathcal{E}^{\approx} and terminates.

Example 21

We apply the algorithm to the running example used in this chapter. As shown in Figure 6.1, the example query has four operators with identifiers 1 to 4. Annotation columns with the identifier 0 refer to the table access annotations described in Example 17. Let us now assume, the algorithm has processed the schema alternative S_1 , which does not apply any attribute replacements on the query. The set of approximate explanations \mathcal{E}^{\approx} already holds a single explanation $\{2\}$, which points to the filter in the example query. The algorithm currently processes schema alternative S_2 , which replaces the *address1* attribute with *address2*. Therefore, Breadcrumb initializes E_{S_2} to $E_{S_2} = \{1\}$, in which the identifier 1 represents the flatten operator that is reparameterized on the *address* attribute. Further, Breadcrumb computes θ_{S_2} and RA as follows:

$$\begin{aligned}
\theta_{S_1} = & (c_S2_0 = 1) \vee (c_S2_0 = \perp) && \vee \\
& (v_S2_1 = 1 \wedge c_S2_1 = 1) \vee (v_S2_1 = \perp \wedge c_S2_1 = \perp) && \vee \\
& (v_S2_2 = 1 \wedge c_S2_2 = 1) \vee (v_S2_2 = \perp \wedge c_S2_2 = \perp) && \vee \\
& (v_S2_3 = 1 \wedge c_S2_3 = 1) \vee (v_S2_3 = \perp \wedge c_S2_3 = \perp) && \vee \\
& (v_S2_4 = 1 \wedge c_S2_4 = 1) \vee (v_S2_4 = \perp \wedge c_S2_4 = \perp)
\end{aligned}$$

$$RA = \{r_S2_1, r_S2_2, r_S2_3, r_S2_4\}$$

When *Breadcrumb* has conducted the selection with θ_{S_1} , the projection on RA , and the deduplication on our reduced running example, it obtains the $R_{S_2}^A$ in Table 6.13. It holds one tuple with 1-values in all retained attributes.

r_S2_1	r_S2_2	r_S2_3	r_S2_4
1	1	1	1

Table 6.13: *retained* tuples in $R_{S_2}^A$ after selection, projection, and deduplication

This tuple originates from the NY tuple 992 in Table 6.12. It holds two Sues in the *nList_S2* attribute with identifiers 715 and 716. Without deduplication, $R_{S_2}^A$ would contain two entries with 1-values in all retained attributes because of the two nested Sues. The deduplication ensures that each explanation is unique in the context of a schema alternative.

Breadcrumb extracts the operators from $R_{S_2}^A$. In this example, *Breadcrumb* only adds the empty set \emptyset to $\mathcal{E}_{S_2}^{\approx}$ because it processes just one tuple and this tuple holds only 1-values. If, for instance, $R_{S_2}^A$ contained another tuple with $r_S2_4 = 0$, *Breadcrumb* would also add the set $\{4\}$ to $\mathcal{E}_{S_2}^{\approx}$. In this example, however, *Breadcrumb* unions $E_{S_2} = \{1\}$ with \emptyset to obtain $E_{S_2}' = \{1\}$. It adds E_{S_2}' to $\mathcal{E}_{S_2}'^{\approx}$ before returning from the loop that iterates through the schema alternatives. *Breadcrumb* unions $\mathcal{E}_{S_2}'^{\approx}$ with $\mathcal{E}^{\approx} = \{\{2\}\}$. Now, \mathcal{E}^{\approx} contains the two explanations $E_1 = \{2\}$ and $E_2 = \{1\}$: $\mathcal{E}^{\approx} = \{\{2\}, \{1\}\}$. Since *Breadcrumb* has computed the explanations, now, it advances to the pruning and sorting.

Breadcrumb prunes the approximate explanations \mathcal{E}^{\approx} based on upper and lower bounds (*UB* and *LB*) for side-effects. These bounds are independent of actual operator reparameterizations. They describe the maximum and the minimum number of side-effects of any possible actual operator

reparameterization. More precisely, there exist no actual operator reparameterizations that generate more side-effects than defined by the upper bound. Similarly, there exist no actual operator reparameterizations that generate fewer side-effects than defined by the lower bound.

Both bounds exclusively consider top-level tuples since computing the tree-edit distance of unordered trees in the nested result data is NP-hard [ZSS92]. Furthermore, both bounds consider tuples that *any* operator reparameterization in an explanation adds (Δ^+) or removes (Δ^-) from the original query result $\llbracket Q \rrbracket$. Breadcrumb considers the original schema alternative, consistently denoted as S_1 , differently than the other schema alternatives.

The upper bounds are computed as follows. For S_1 , $UB(\Delta^+)$ equals the number of valid top-level tuples in the result that have at least one retained flag set to 0 for one of the explanation's operators. For all other schema alternatives $S_i, i \neq 1$, $UB(\Delta^+)$ is the number of valid top-level tuples that can become part of the result under S_i but do not have all the *retained* and *valid* flags set to 1 under S_1 . The number of removed tuples $UB(\Delta^-)$ equals $|\llbracket Q \rrbracket|$ minus the number of valid top-level tuples under the considered alternative that match a tuple with all *retained* and *valid* flags set to 1 under S_1 .

Breadcrumb generally estimates the lower bounds

$$LB(\Delta^+) = \max(\text{number of valid and retained tuples} - |\llbracket Q \rrbracket|, 0)$$

and

$$LB(\Delta^-) = \max(|\llbracket Q \rrbracket| - \text{number of valid and retained tuples}, 0).$$

Intuitively these bounds describe top-level tuples that any reparameterization based on an explanation adds or removes to or from the actual query result $\llbracket Q \rrbracket$. Further, Breadcrumb sets lower bounds of explanations involving a selection or join to 0 since it does not have the information whether a reparameterization different from the “full relaxation” of the operator may avoid the side-effects.

We leave algorithms that compute tighter bounds to future work. Finally, Breadcrumb orders the explanations based on the partial order defined in Definition 6.8 leveraging the lower bounds, i.e., $LB = LB(\Delta^+) + LB(\Delta^-)$.

Example 22

Recall that our reduced running example yields a query result that holds only one top-level tuple with the *city* LA and two *firstnames* Sue in the nested relation *nList*. Therefore, Breadcrumb computes the lower and upper bounds for the explanations E_1 as follows:

- $UB_{E_1}(\Delta^+) = 2$, because a reparameterization may add two tuples representing NY and SF
- $UB_{E_1}(\Delta^-) = 0$, because the LA tuple appears in all valid reparameterizations of explanation E_1
- $LB_{E_1}(\Delta^+) = 0$, because the explanation contains a selection
- $LB_{E_1}(\Delta^-) = 0$, because the explanation contains a selection

The upper and lower bounds for E_2 are:

- $UB_{E_2}(\Delta^+) = 1$, because the exclusive reparameterization of the flatten operator yields at most one NY tuple in the result
- $UB_{E_2}(\Delta^-) = 1$, because the LA tuple does not appear in the result under any reparameterization of E_2
- $LB_{E_2}(\Delta^+) = 0$, because the size of the query result is 1 and the number of valid and retained tuples from the unmodified query is 0 under all reparameterizations of E_2
- $LB_{E_2}(\Delta^-) = 1$, because the query result contains one tuple that is not valid and retained under S_1

Even without these bounds, Breadcrumb does not prune E_1 or E_2 from \mathcal{E}^\approx , because the operators in E_1 and E_2 distinguish from each other. However, Breadcrumb considers the side-effects for the final ordering. Since E_1 or E_2 contain just one operator, it sums up lower bounds' values. For E_1 , it is 0. For

E_2 , it is 1. Thus, in this example, Breadcrumb ranks E_1 higher than E_2 . It returns $\mathcal{E}^\approx = \{\{2\}, \{1\}\}$.

6.3.5 Discussion

Breadcrumb may miss or over-approximate explanations. In this section, we describe the limitations of Breadcrumb regarding the approximated explanations. We further describe the runtime complexity of computing the explanations and provide reasoning why Breadcrumb scales to large datasets in big data analytics systems.

Approximate explanations. Breadcrumb guarantees that any returned explanation is correct, i.e., each explanation yields at least one successful reparameterization. Due to the loose bounds on side-effects, not all explanations are guaranteed to yield minimal successful reparameterizations.

Furthermore, Breadcrumb may miss some explanations due to its heuristic nature. Essentially, the proposed algorithm applies the following systematic optimizations for efficiency that cause certain cases not to be accurately covered: First, it considers only equi-joins and does not model a reparameterization to theta-joins. As a consequence, it avoids computing the cross-product from the two input relations and enumerating all possible outputs of join reparameterizations. If such a reparameterization was part of an explanation, our algorithm misses it. Second, Breadcrumb exclusively considers reparameterizations for the selection, join, and flatten operators that are less selective than the original operators after applying each schema alternative. Hence, it misses explanations that require a more restrictive selection condition, join type, or flatten type. Third, for aggregations, Breadcrumb generally does not consider changing the aggregation function. Further, it does not trace the result for different subsets of the aggregation input data. Thus, Breadcrumb may over-approximate or miss explanations for queries in which selections, equi-joins, or flatten operators precede aggregations.

Runtime complexity. We briefly sketch Breadcrumb's runtime complexity given a query Q with q operators. The schema backtracing step first applies a schema matching as described in Section 4.3.1 with the complexity

described in Section 4.3.3. The schema backtracing then iterates through the operators in the query starting at the last operator. For each operator, it has to consider p path manipulations. Thus, the schema backtracing has an overall complexity of $O(q \cdot p)$. However note, that the path manipulations are bound by the maximum number of attributes in the input and output schema of each operator. The exact number highly varies between the operators. For instance, a filter does not conduct any path manipulations. A renaming conducts one manipulation for each top-level attribute in the data. In practice, p typically is much smaller than the number of attributes in the schema.

In the second step, the algorithm searches for schema alternatives. Finding the schema alternatives involves comparing all provided attribute alternatives with each other. Let us assume n attributes referenced in the query have at most a alternatives each, which is bound by the number of attributes in each input relation. Then, computing all possible alternatives has exponential complexity $O(a^n)$. Note that the exponential complexity is on the schema, not on the data. Hence, it is practically feasible to compute the schema alternatives.

The data tracing step computes annotations for each operator and each schema alternative. The number of schema alternatives is $s = |\mathcal{S}| \leq a^n$. Remember that Breadcrumb annotates each top-level tuple in the output of each operator in Q . Let the maximum number of top-level output tuples that any of the operators in Q produce be d . For each top-level tuple, the algorithm generates a unique identifier. Furthermore, it creates a consistent, retained, and valid annotation. Hence, the algorithm produces at most $q \cdot d \cdot (1 + s \cdot 3)$ annotations. Computing the annotations for each top-level is in constant time for the retained, and valid attribute for all operators but the relation nesting operator. The relation nesting operator needs to check all valid annotations of the input tuples that contribute to each output tuple. The consistent annotations require the execution of the tree-pattern matching's data matching step. It has a complexity of $O(m)$ for each d (Section 4.3.3), where m is the number of attribute nodes referenced in the why-not question. Thus, the overall time complexity for data tracing per operator is $O(m \cdot s \cdot d)$.

Since s and m are typically orders of magnitude smaller than d and each operator in the pipeline has at least a complexity of $O(d)$, the data tracing procedures do not impose practically infeasible runtime overhead. However, the size of the annotations grows with each operator in Q by $d \cdot (1 + s \cdot 3)$, as described above. Thus, the data that has to be shuffled across the network permanently grows. Therefore, in practice, the data tracing procedures have a significant impact on the runtime.

Computing the explanations requires Breadcrumb to check all annotation fields $f = q \cdot d \cdot (1 + s \cdot 3)$. Therefore, the runtime complexity to compute the explanations is in $O(f)$. Next, the algorithm computes the lower and upper bounds for each explanation. The algorithm potentially finds each possible operator combination as an explanation. That leads to a total number of $e = 2^q$ possible explanations, where $e < d$ since one top-level tuple can produce at most one explanation. Computing the lower and upper bounds for each explanation only depends on the number of top-level tuples d . Thus, this part has a complexity of $O(e \cdot d)$. Eventually, the algorithm sorts the e explanations according to the partial order in Definition 6.7. Efficient sorting has a complexity of $O(e \cdot \log(e))$. Replacing e with $e = 2^q$ yields $O(2^q \cdot \log(2^q)) = O(2^q \cdot q)$. However, in practice, the number of explanations is significantly lower than e . It usually does not exceed two-digit figures. Therefore, the runtime of this compute step is dominated by finding the explanations.

Scalability. Part of our Contribution (4) is that Breadcrumb scales to large datasets in distributed big data analytics systems. Here, we discuss why Breadcrumb’s design allows the algorithm to scale. Recall that Breadcrumb’s first two steps, the schema backtracing and schema alternative computation, exclusively run on the schema. The schema typically is orders of magnitude smaller than the data. Thus, distributing these steps will not have a significant impact on Breadcrumb’s runtime. However, the final two steps, data tracing and explanation computing process the data. Therefore, we focus on distributing the execution of these two steps.

As the pseudo-code of Breadcrumb’s tracing procedures in Section 6.3.3 illustrate, the procedures iterate over each schema alternative and top-level

output tuple to compute the annotations. Breadcrumb can parallelize and distribute the annotation computation since computing them only requires access to exactly the tuple for which the annotations are computed. Furthermore, Breadcrumb can distribute the computation of the approximate explanations described in Section 6.3.4 by leveraging the distributed operators of the big data analytics system. For each schema alternative, it applies a selection, projection, and distinct operator. The big data analytics system offers highly optimized, distributed implementations for these operators. Hence, Breadcrumb distributes the explanation computation by utilizing the underlying big data system. In Chapter 8, we confirm that Breadcrumb scales to large datasets with an increasing number of schema alternatives. Here, we conclude this chapter with a summary.

6.4 Summary

In this chapter, we have formally introduced the why-not question that requests expected, but missing data in the result. To find query-based explanations for the missing data in the question, we have introduced reparameterizations. They are successful if they produce the missing tuple described in the why-not question. We have further introduced a partial order over the successful reparameterizations to find minimal successful reparameterizations. They keep the impact on the query and result at a minimum. Hence, the operators in these minimal successful reparameterizations form the explanations for the missing answers.

We have described why computing these explanations is infeasible. That is why we have proposed the heuristic Breadcrumb algorithm that computes approximate explanations in four steps. Its schema backtracing step traces the schema back from the output to the input. Its consecutive schema alternative step computes schema alternatives. Based on these schema alternatives, the data tracing step annotates the data during processing. Based on the previously annotated data, the final step computes the explanation. We have discussed that Breadcrumb approximates the formally defined explana-

tions, has reasonable theoretical runtime complexity, and scales on big data analytics systems.

All in all, we contributed a novel approach to identify query-based explanations for missing data based on reparameterizations that extends existing, provenance-based approaches to provide more comprehensive explanations than any previous approach can provide. While previous approaches were limited to flat relational data, our approach also supports nested data. Hence, it supports a larger variety of data input formats that are particularly relevant in big data analytics systems. Our novel Breadcrumb algorithm computes the explanations leveraging a distributed big data system. It scales to datasets of practically relevant size as we show in the evaluation chapter (Chapter 8), while other existing solutions have only been evaluated on floppy-disk-sized datasets. In the next chapter, we discuss Breadcrumb's, Pebble's, and the tree-pattern matching algorithm's implementation.

IMPLEMENTATION

The concepts and algorithms described in Chapters 4 to 6 are generally applicable to big data analytics systems that support data-parallel batch-processing, such as Apache Spark, Flink, or Hive. This chapter describes their implementation and extends the implementation sections in our published papers [DH20a; DH20b; DLHG21a; DLHG21b].

As part of Contribution (5), we implement them in Apache Spark [ZXW+16] to demonstrate their feasibility and evaluate their scalability. For that purpose, we briefly describe the internals of Spark needed to understand our implementation before we discuss our implementation details.

We implement our concepts on top of Spark’s Dataframe API [AXL+15]. This API processes dataframes. These are distributed data collections with bag semantics that support nested data collections. These nested data collections are implemented as nested arrays, which provide a unique position inside the array for each element. We leverage this position as unique identifiers for nested tuples in the tree-pattern matching, the Pebble, and the Breadcrumb implementation. Since dataframes offer bag semantics and provide unique positions for nested elements they match the formal data model introduced in Section 3.1.

To manipulate the dataframes, Spark offers transformations. They resemble SQL operators, such as selections, projections, and joins. Further transformations manipulate nested data, such as flatten and nesting. Together, these transformations closely correspond to the operators in the execution model introduced in Section 3.2. Therefore, Spark’s Dataframe API meets the requirements to implement our concepts.

Transformations provide the blueprint to manipulate the dataframes. Combining them leads to a blueprint of a query pipeline that resembles the query definition in SQL. Before Spark starts the pipeline execution it analyzes the processing pipeline definition and optimizes the pipeline similarly to relational database systems that optimize SQL queries. Spark conducts these optimizations exclusively based on the data’s schema and optional metadata, when available. We call this phase the planning phase. Since the plan typically is orders of magnitudes smaller than the data to be processed, Spark executes the planning on a single compute node.

Spark’s actions trigger the actual execution of the query pipelines. They initiate the execution phase. During this phase, Spark reads the input data to compute the query result on the distributed compute nodes of the big data analytics systems. Common examples for actions are collect (which provides the execution result in an in-memory array), write (which writes the computed result into files on a filesystem), or count (which returns the number of top-level tuples in the result).

After this brief introduction of the basic Spark concepts and terminology, we focus on the implementation of our three algorithms. More precisely, we describe how we leverage Spark’s Dataframe API to implement the tree-pattern matching prototype (Section 7.1), the Pebble prototype, and the Breadcrumb prototype. We describe the latter ones by comparing their implementations (Section 7.2). We conclude the chapter with a discussion on implementation decisions (Section 7.3).

7.1 Tree-pattern matching implementation

We implement the distributed tree-pattern matching algorithm as a novel transformation on Spark’s dataframes. This transformation seamlessly integrates with the other transformations. Developers can use and arbitrarily combine it with other transformations when defining query pipelines. The transformation integrates into Spark’s pipeline planning and execution phases to achieve optimal distribution and scalability on the big data analytics system.

As summarized in Algorithm 1, the tree-pattern matching algorithm has two major phases. In the schema-matching phase, the algorithm computes the schema-matches and in the data-matching phase, the algorithm computes matches on top-level tuples in the data. We describe the implementation of the schema-matching phase before we explain the implementation of the data-matching phase.

7.1.1 Schema matching

As described in Section 4.3.1, the schema matching phase computes schema-matches given the schema of the input dataframe and the tree-pattern. The schema-matching phase is fully implemented into Spark’s planning phase. When Spark analyzes a pipeline definition containing a tree-pattern matching transformation, the transformation’s implementation computes the schema-matches during the planning phase based on the data’s schema. For that purpose, it implements the DeweyIDs [LLCC05] that assign each schema node a unique identifier as strings, the index $I_{\mathcal{R}}$ that maps the schema node labels to DeweyIDs as a hash-map, and the tree-patterns T and schema-matches $M \in \mathcal{M}$ as trees. While more efficient data structures exist to realize the mentioned parts, the runtime performance is sufficient for our prototypical implementation, particularly because our approach exclusively conducts the matching on the schema.

The schema-matching further prepares the schema-matches $M \in \mathcal{M}$ for distributed processing. It applies a custom serialization that generates an

array of the nodes in each schema-match M . This array becomes an input parameter of the data matching implementation and is broadcasted to all compute nodes in the cluster. This implementation integrates smoothly into Spark's data structures.

In a final step, the schema-matching implementation registers a user-defined function in Spark's execution engine and integrates it into the pipeline execution plan. This user-defined function contains the code for the data matching phase.

7.1.2 Data matching

Recall from Section 4.3.2 that the data matching phase takes the schema-matches and the data as input to return the final tree-pattern matches. The data matching phase is implemented as a user-defined function (UDF) integrated into the pipeline plan during the schema matching phase. In the execution phase, Spark leverages the built-in *map* transformation that applies the UDF for each top-level tuple in the input data. The UDF first deserializes the schema-matches $M \in \mathcal{M}$ so that the UDF can recursively and simultaneously iterate through the nodes in M and the data in each top-level tuple. During this iteration, it verifies the nodes' constraints against the data in the top-level tuple to find matches.

Recall from Section 4.3.2 that the implementation of the data matching phase has either of three types. The first type extends the input relation with an attribute that holds boolean values. They indicate whether the top-level tuple matches the tree-pattern. It is described in detail in Section 4.3.2. The second type returns a nested relation of the type defined by some particular output nodes in the tree-pattern. The third type returns a relation that holds the exact paths the tree-pattern has matched on. While the former two are exposed to the application via the API, the latter one is exclusively used by the Pebble module to trace the structural provenance.

The second and third types distinguish from the first type in the number of output tuples generated for each input tuple. The operator potentially yields multiple output tuples per input tuple, because a tree-pattern may

match multiple times on a single input tuple. Therefore, Spark leverages the built-in *flatMap* transformation for the second and third type instead of the *map* transformation for the first type to incorporate the varying number of output tuples.

For the second type, the algorithm further restructures the input so that the output matches the selected output nodes in the tree-pattern. Therefore, it creates a temporary buffer for the output nodes. While iterating through the data, it does not only verify the constraints but also copies the data matching the output nodes into the buffer. Once the algorithm has filled all output nodes in the buffer and has verified that all constraints are satisfied, it emits the buffered content as a top-level output tuple.

For the third type, the algorithm computes an array for each match. This array contains one entry for each path to the matching data. These paths include the exact positions of the data in the input data, i.e. the position of elements in the nested relations. For that purpose, the algorithm records the exact paths while iterating through the data for verification purposes. It buffers all matching paths until it has verified that the data in the top-level tuple matches the tree-pattern. Then, it emits the array together with the input tuple's identifier as a newly created top-level output tuple.

In summary, we integrate the tree-pattern matching algorithm into Spark's query planning and execution phase. The schema-matching phase integrates in Spark's query planning phase and the data matching phase in Spark's execution phase. Since we have split the tree-pattern matching algorithm in these two dedicated phases, it can evaluate each top-level tuple in the input data independently of all other tuples. Hence, it can leverage Spark's built-in *map* and *flatMap* function to distribute the data matching phase. They are optimized for distributed execution. Thus, our tree-pattern matching algorithm scales with increasing data sizes and compute resources. Pebble's and Breadcrumb's implementations utilize the tree-pattern matching algorithm to efficiently identify and trace nested data. We describe their implementations next.

7.2 Pebble and Breadcrumb implementation

Like the tree-pattern matching implementation, Pebble's and Breadcrumb's implementations also extend Apache Spark. While Pebble traces the provenance of existing data, Breadcrumb traces the provenance of missing data. Both trace provenance and, thus, share common implementation concepts. Therefore, we jointly describe their implementation. We focus on two aspects of their implementation. The first aspect covers the implementation details needed to understand how we extended Spark with Pebble and Breadcrumb. More precisely, we first describe the annotations implementation in Spark's dataframes. Pebble and Breadcrumb rewrite Spark's transformations to update the annotations. We describe the different approaches that Pebble and Breadcrumb implement to rewrite the operators and give reason why they pursue different approaches. The rewrite is a prerequisite to querying the explanations. Both algorithms provide different means to query the provenance. We describe and reason about their implementations. We further elaborate on Breadcrumb's schema alternative implementation. The second aspect addresses optimizations that let Pebble and Breadcrumb scale to large datasets. While both algorithms are designed to run in distributed big data analytics systems they need further optimizations in Spark to scale to large datasets. We describe these optimizations at the end of this section. We start the section with the annotations implementation, next.

7.2.1 Annotations

As described in Chapters 5 and 6 the algorithms leverage annotations during query pipeline processing to compute explanations. In both algorithm implementations, these annotations are stored in additional attributes in each top-level tuple. In Spark's dataframes, they are then available as additional columns. These columns typically are of primitive type (i.e., long type for identifiers or boolean type for other annotations).

Notable exceptions are the annotation columns created for the relation nesting and aggregation transformations since they are of nested relation

type. In Spark, nested relations are implemented as arrays. Thus, each tuple in the nested relation has a position and it is possible to access every tuple by its position. For the two mentioned transformations, the algorithms collect nested relations of annotation tuples which encapsulate the annotations added by preceding transformations. For the relation nesting transformation, the algorithms further ensure that the annotation tuple has the same position in the nested annotation relation as the corresponding tuple in the newly created nested collection. This implicit correlation by position allows for associating the annotations with the correct nested tuple.

Similarly, Pebble and Breadcrumb leverage the position information when unnesting a nested relation. They record the position as part of the unique identifier for each unnested tuple from the nested relation. Therefore, they can correctly track each unnested value back to its nested origins.

Recall that Pebble's annotations only consist of tuple identifiers. However, Breadcrumb's annotations further record *valid*, *retained*, and *consistent* annotations for each schema alternative. Thus, Breadcrumb's annotations are more comprehensive.

In summary, Pebble's and Breadcrumb's tracing procedures permanently update and manipulate the annotations. To handle the annotations for nested data, they leverage the positions implicitly provided in Spark. Next, we describe how these tracing procedures are wrapped around each transformation to achieve their purpose.

7.2.2 Transformations

In Spark, transformations are implemented as Scala classes that are instantiated and parameterized for each occurrence in a query pipeline. These classes implement an *execute* method that defines the execution behavior. Both, Pebble and Breadcrumb extend these transformations to compute the previously described annotations. However, their implementations significantly differ from each other because Pebble stores annotations in separate relations alongside the query execution and Breadcrumb propagates the

annotations through the query pipeline. We describe Pebble’s approach first and Breadcrumb’s approach afterward.

Pebble directly derives an extended class from each transformation class. This class extends the transformation-specific behavior of the original class with the annotation tracing. For that purpose, Pebble overrides the *execute* method for each transformation. While the actual implementation of each transformation extension depends on the transformation, their extended *execute* methods share a common structure. They all have a preprocessing step, the transformation step, and the postprocessing step.

During the preprocessing, Pebble prepares the transformation execution. It ensures that the consecutive transformation does not fail because some preconditions are not met and the annotations are not removed or unintentionally modified during the actual transformation execution step. For instance, the union operator requires dataframes with matching schemata. While the input schemata for the union operator without annotations match, they may not match for the annotated schemata. Thus, Pebble aligns the schemata before executing the union operator. Furthermore, Pebble manipulates the list of projection attributes prior to the projection’s execution so that the projection does not remove the annotations.

The transformation step implements the transformation-specific behavior. Pebble applies Spark’s transformation execution implementation in this step. However, the parameterizations may have been modified in the previous step.

During the postprocessing step, Pebble adds and updates the annotations. In case the transformation yields at most one top-level tuple for each top-level output tuple such as the selection or projection transformation, Pebble simply propagates the provenance annotations in the post-processing step. When a transformation such as the join or the relation nesting transformation combines multiple top-level input tuples into a new output tuple, Pebble assigns each top-level tuple a new unique identifier during the post-processing. In either case, Pebble only propagates the most recently created provenance annotations and caches the mapping between the input identifiers and the

new output identifier. Hence, Pebble keeps the execution overhead during query pipeline execution low.

Breadcrumb pursues another implementation approach to extend each transformation with annotation tracking. After defining and before the query pipeline execution, it rewrites the query plan to track the annotations. Breadcrumb recursively iterates through the transformations in the query and rewrites each transformation individually starting at the input and finishing at the output.

During the recursion, it manipulates each transformation in the plan to (i) preserve the provenance annotations added by previous transformations, (ii) add new transformation-specific annotations, and (iii) retains tuples in the transformation's output that may be removed by the transformation and may contribute to an explanation. Breadcrumb propagates all annotations through the entire query pipeline. For instance, when a pipeline has four operators, such as our running example pipeline, the query result contains annotations from all four operators rather than only the final tuple identifier. Furthermore, Breadcrumb rewrites the transformations such that it records the necessary *valid*, *retained*, and *consistent* annotations for the transformation. For instance, it leverages the filter condition of a selection transformation to compute the *retained* annotation value. Additionally, Breadcrumb may replace transformations with other transformations to retain tuples that may contribute to the explanation. For example, it replaces the selection transformation with a projection that propagates all input tuples to the output. It extends the projection such that the projection generates the mentioned annotations.

As mentioned in the previous Section 7.2.1, Breadcrumb has to compute the *valid*, *retained*, and *consistent* annotations for each schema alternative. For that purpose, it parses all attribute references in the transformation parameters and rewrites the parameters to take all schema alternatives into account. For instance, Breadcrumb parses the attribute list in the projection and, for each attribute in the list and each schema alternative, it adds the alternative attribute to the projection list. Similarly, Breadcrumb parses the filter condition of a selection transformation and replaces attribute references

with their alternatives to compute the *retained* annotations for each schema alternative.

In summary, Pebble and Breadcrumb apply different implementation approaches to make the transformations collect the annotations. While we found Breadcrumb's approach of extending the transformations to collect annotations easier to implement, it was not a viable approach for Pebble. Recall that Pebble caches as many annotations as possible alongside query execution and only propagates the latest annotations. Rewriting the plan does not provide the flexibility to implement the caching mechanism, because Spark assumes that transformations in the plan always have exactly one output relation. Hence, plan rewriting does not allow for caching the annotations into a separate relation alongside the query execution. In contrast, Breadcrumb propagates all annotations through the query. To account for schema alternatives, Breadcrumb further computes significantly more annotations than Pebble. To this point, it may not be obvious, yet, why Pebble and Breadcrumb distinguish in these implementation details. Pebble and Breadcrumb leverage different computation methods, which we discuss next.

7.2.3 Lazy and eager explanation computation

Pebble and Breadcrumb only compute the explanations when developers query them leveraging novel transformations that extend Spark's dataframes and compute the requested explanations. The general query flow is essentially the same for both explanations and has the following steps. First, the developers define a query pipeline. Then, they trigger one of Spark's actions to execute the pipeline and obtain a result. Next, they observe an unexpected result and leverage Pebble to obtain an explanation for an existing result, or Breadcrumb to obtain an explanation for a missing result.

Pebble and Breadcrumb integrate into different steps of the introduced flow. Pebble *eagerly* collects the annotations when the developers execute the pipeline for the first time to obtain the result. Pebble leverages these annotations once it computes an explanation. In contrast, Breadcrumb does

not impose any footprint on the execution that computes the query pipeline result. It *lazily* collects the annotations right after the developers request the explanation.

We made these implementation decisions to minimize the overall execution overhead. Pebble only adds lightweight annotations to each processed dataframe, for it adds at most three annotation attributes to each intermediate result in the query pipeline. Furthermore, it only annotates tuples in the intermediate and final results that actually occur in the respective result. Therefore, the overall execution overhead is low, and collecting the annotations during the initial pipeline execution is reasonable. On the contrary, Breadcrumb computes the identifier attribute and at least three further annotation attributes for each schema alternative. It further has to preserve attributes that serve as an alternative to the original attributes. Additionally, Breadcrumb has to consider tuples that the unmodified execution pipeline would have removed from the result. Therefore, Breadcrumb potentially causes large intermediate and final results that are expensive to compute. Consequently, Breadcrumb generally imposes such a high execution overhead on the initial pipeline execution that it is unjustifiable to eagerly compute annotations that may not be used, e.g., when the obtained result meets the developers' expectations.

During explanation computation, Pebble mainly needs to join the identifier annotations of the intermediate results. While joins typically are more compute intense than selections or projections, Pebble does not need to re-execute the entire query pipeline. In contrast, Breadcrumb has to re-execute the query pipeline from scratch and compute the annotations for all schema alternatives simultaneously during the re-execution. If Breadcrumb then cached the annotations of intermediate results like Pebble it would have to join them after the re-execution. That imposes even more overhead than propagating all annotations. Therefore, Breadcrumb propagates the annotations.

In fact, we have implemented the latter approach in Pebble to compare eager provenance capture with lazy provenance capture. The experiments reported in the next chapter reassured us, that propagating the annotations

in Breadcrumb is the most efficient way to compute the explanations lazily. However, this is not the only design choice to make Pebble and Breadcrumb scale on large datasets in distributed computing clusters. In the following subsection, we highlight two of the implementation optimizations that make the algorithms scale.

7.2.4 Rewrite optimizations and limitations

To make Pebble and Breadcrumb efficient and scalable, we conduct multiple optimizations in their implementations. While Pebble’s implementation also profits from the reported optimizations in Titian [IST+15], Breadcrumb’s implementation required novel implementation optimizations particularly due to the introduction of schema alternatives. Recall that Breadcrumb basically executes multiple queries simultaneously, (i.e., one query per schema alternative). Naive implementations of the relation flatten transformation and the relation nesting transformation do not scale to the extent needed for large datasets. Therefore, we highlight our optimizations on these two transformations. Afterward, we describe implementation limitations on the join transformation imposed on Breadcrumb by Spark.

Flatten. Without schema alternatives, the relation flatten transformation unnests a nested relation and merges the attributes of the nested tuple with the attributes of the tuple that holds the nested relation. The naive approach to implement the flatten transformation for schema alternatives is to apply the flatten on each nested relation alternative individually and union their results in a consecutive step. This approach has two caveats. First, Spark computes the input of the flatten transformation from scratch for each nested relation alternative. Breadcrumb could leverage Spark’s caching mechanisms to overcome this caveat. Second, the result of the naive implementation is blown up. It creates one row for each combination of the top-level tuple holding the nested relation and the tuple in the nested relation for each alternative nested relation. As a consequence, in the transformation’s output, each combination is represented in its own top-level tuple. The result would

be more compact if each tuple holds the combination of one input top-level tuple and nested tuple per alternative.

Breadcrumb adapts concepts from functional programming to overcome both caveats of the naive flatten implementation. In brief, Breadcrumb emulates the zip function and the flatmap function to simultaneously flatten alternative nested relations. In detail, Breadcrumb extends the flatten transformation as follows. First, it computes the maximum number n of elements of all alternative nested relations for each input top-level tuple. For each of the input tuples, Breadcrumb iterates from 0 to n and creates one output tuple for each iteration i (resembles flatmap). Thus, it generates n output tuples for each input tuple. For iteration i , the output tuple holds the attributes and values of the top-level tuple plus the attributes and values from the i th tuple of each alternative nested relation (resembles iterating through zipped relations). If one alternative relation holds less than i elements, Breadcrumb sets the corresponding attribute values to null and the *valid* annotation for that alternative to false. Eventually, Breadcrumb creates only n output tuples for each input tuple instead of n times the number of schema alternatives. Furthermore, Breadcrumb does not require to union intermediate results obtained for individual schema alternatives because each output tuple considers all schema alternatives. Therefore, Breadcrumb's implementation of the relation flatten transformation overcomes the two caveats and scales to a high number of alternatives and a large number of top-level tuples as the experiments in the next chapter validate.

Relation nesting/aggregation. In Spark, our formal relation nesting and aggregation transformations map both to the groupby and aggregation transformation known from SQL. They only distinguish in the aggregation function. Extending this SQL-like transformation is particularly challenging when considering schema alternatives since the grouping attributes may differ between alternatives. The naive implementation of the groupby/aggregation transformation resembles the naive implementation of the flatten transformation. Our extended transformation would compute an individual result for each alternative and then union these partial results in a consecutive step. It has the same caveats as the naive flatten implemen-

tation. Additionally, this transformation requires shuffling the data across the compute nodes in the distributed cluster. Thus, computing partial results is particularly compute-intensive and time-consuming. The following optimized implementation addresses these caveats.

Breadcrumb replaces the single groupby/aggregation transformation with one expand transformation and two succeeding groupby/aggregation transformations. The expand transformation duplicates the top-level tuples in the input for each schema alternative. The first groupby/aggregation transformation computes the nested relations or aggregated values for each schema alternative. The second aggregation merges the results from each schema alternative into a single output relation. The result relation then holds the same grouping attributes for all schema alternatives and distinct attributes for each alternative's aggregation attribute. As a consequence, the rewritten groupby/aggregation transformation always requires just two shufflings regardless of the number of schema alternatives. Furthermore, it avoids the recomputation from scratch to obtain intermediate results and provides a compact representation of the result which positively impacts the performance of succeeding transformations. It has a disadvantage over the naive implementation, though. Duplicating the tuples is memory intensive. If the compute cluster has insufficient memory to keep the duplicates in memory, the computation significantly slows down due to necessary disk I/O.

Join. We have experienced another performance bottleneck when implementing Breadcrumb's join transformation extension. Breadcrumb rewrites each join to a full outer join to ensure that it correctly tracks all tuples. When a schema alternative substitutes an attribute in the join condition with another attribute, Breadcrumb copies the join condition, replaces the attribute, and disjunctively combines the condition with the original condition. The combination of the full outer join and the disjunction in the join condition make Spark perform a cross-product over the join's input relations. Consequently, the execution time grows infeasibly high on large datasets. Therefore, Breadcrumb currently does not support alternative join conditions on large datasets. We are optimistic that improvements in Spark will overcome this shortcoming in the near future.

Based on the implementation hints provided in this section, one can understand why Pebble and Breadcrumb scale to large datasets in distributed big data analytics systems. Next, we summarize this chapter.

7.3 Summary

In this chapter, we have described implementation details on the tree-pattern matching algorithm, Pebble, and Breadcrumb. They are relevant to process nested data in a scalable fashion on distributed systems as stated in Contribution (5). All algorithms are implemented as standalone Scala libraries that are binary compatible with multiple Spark versions (i.e., Spark 2.2 - 2.4). Furthermore, Breadcrumb is open-sourced and available on Github¹. We leverage the libraries in the next chapter to assess the algorithms' scalability on large datasets.

¹<https://github.com/UniStuttgart-DataEngineering/breadcrumb>

EVALUATION

In this chapter, we experimentally evaluate the tree-pattern matching algorithm, Pebble, and Breadcrumb. In Contributions (2) to (5), we claim that the algorithms scale to large datasets on distributed big data analytics systems. In this chapter, we present experiments that underline these claims. Furthermore, we show that the tree-pattern matching operator that implements our tree-pattern matching algorithm has the potential to significantly reduce the query pipeline complexity. We also illustrate that Breadcrumb provides such comprehensive explanations that it enables novel applications. Additionally, we compare Breadcrumb’s explanations with the explanations of other state-of-the-art query-based explanations to show that Breadcrumb outperforms them in multiple aspects. For that purpose, we first introduce the hardware setup used for the evaluation (Section 8.1). Then, we describe the three evaluation datasets (Section 8.1.2). To effectively evaluate the algorithms, it is necessary to employ dedicated query workloads for each algorithm. Therefore, we describe the query workload at the beginning of each algorithm’s section. We start with the tree-pattern matching algorithm (Section 8.2), continue with Pebble (Section 8.3), and finish with Bread-

crumb (Section 8.4). We conclude this chapter with a general discussion on the evaluation results.

8.1 Evaluation setup

Before we present evaluation results, we describe the common properties of all conducted experiments. They run on the same compute clusters as described next. Furthermore, they leverage the same datasets, which we describe after the cluster setup. Additionally, all reported runtimes are averaged over five consecutive runs embedded in an additional warm-up and tear-down run.

8.1.1 Cluster setup

We conduct all experiments on either of two clusters if not mentioned otherwise. We summarize their configurations in Table 8.1. Both clusters have one dedicated management node. Furthermore, cluster 1 has three physical compute nodes with overall $3 \cdot 8 = 24$ compute threads, $3 \cdot 244 = 732$ GB memory, and 4TB SSD storage. Cluster 2 has more compute resources. It comes with $6 \cdot 20 = 24$ compute threads, $6 \cdot 176 = 1056$ GB memory, and 4TB main memory. While cluster 1 has fewer compute threads, their clock speed of 3.5 GHz is higher than the clock speed of cluster 2. It has a clocking of 2.1 GHz. Both clusters have similar software stacks. They run Ubuntu 18.04 LTS, which has the OpenJDK 1.8 installed. Further, both have Scala 2.11 installed. While cluster 1 runs Hadoop 3.1.0 and Spark 2.3.1, cluster 2 runs Hadoop 3.2.0 and Spark 2.4.0. Both clusters manage resources with Yarn.

The clusters have a similar software stack. However, they have different hardware setups. Thus they have different execution configurations. Cluster 1 runs with 10 executors that have two execution threads and 60 GB main memory, each. Cluster 2 has 50 executors with two execution threads and 20 GB main memory, each.

When we started the research presented in this work, we conducted the experiments on the smaller cluster 1 because cluster 2 did not exist at

Configuration	Cluster 1	Cluster 2
Hardware Configuration		
Management nodes	1	1
Compute nodes	3	6
Compute threads	24	120
CPU clocking (GHz)	3.5	2.1
Main memory (GB)	732	1056
SSD Storage (TB)	4	20
Software Configuration		
Linux	Ubuntu 18.04	Ubuntu 18.04
Java	OpenJDK 1.8	OpenJDK 1.8
Scala	2.11	2.11
Hadoop	3.1.0	3.2.0
Spark	2.3.1	2.4.0
Cluster manager	Yarn	Yarn
Execution Configuration		
Executors	10	50
Threads per executor	2	2
Executor memory	60	16

Table 8.1: Cluster setups of the two Spark clusters used for the evaluation

this time. Once the more powerful cluster 2 was available, we exclusively leveraged this cluster for experiments.

8.1.2 Datasets

We base our evaluations on three nested datasets. The first dataset is a modified, synthetic benchmark dataset. The second and third datasets are real-world datasets. In the following, we describe the key properties of each of the datasets.

Nested TPC-H. We leverage a nested version of the synthetic TPC-H benchmark. The TPC-H dataset holds customers, orders, lineitems, parts, and suppliers in a normalized schema [Tra09]. Since we have designed the algorithms with nested data in mind, we nest the lineitems into the orders as described in [PCW17]. By default, our TPC-H dataset has a scale factor

of 10. It has a size of 10GB and contains roughly 15 million orders, which hold more than 60 million nested lineitems in total.

Twitter. The Twitter dataset contains nested tweets obtained from the public Twitter API. In this real-world dataset, each tweet has up to 1000 attributes arranged in up to eight layers of nesting [WC17]. Among other information, the attributes hold information about the tweeted texts, used hashtags, the tweets' authors, the users mentioned in the tweet, and media links. Furthermore, they hold quoted and retweeted tweets. Many attribute names repeat in the Twitter schema. For instance, the attributes of the tweets' authors highly overlap with the attributes of the users mentioned in the tweets. Similarly, the tweet itself shares a large number of attributes with the quoted and retweeted tweets. The dataset holds up to 130 million tweets at its maximum size. Its size ranges from 100GB to 500GB in increments of 100GB. The default size in the following experiments is 100GB.

DBLP. The DBLP dataset describes computer-science-related publications, conferences, journals, and authors [Ley09]. We have downloaded a single `dblp.xml` file from `dblp.org`, which roughly has size of 2GB. To conduct meaningful experiments on the dataset, we split the records in the XML file by type. Records have one of ten types. In this work, we focus on the five most significant types, i.e. proceedings, inproceedings, journals, articles and, authors. We store each of the types in separate nested relations and scale them in a way that preserves key characteristics in the dataset, such as the average number of inproceedings per proceeding or the average number of authors per article. Each relation has roughly 20 to 50 attributes which have up to three layers of nesting. In total, the dataset holds up to 1.5 billion records at a size of 500GB. Like the Twitter dataset, the DBLP dataset has a size of 100GB to 500GB. The default dataset size also is 100GB.

In the following, we evaluate the tree-pattern matching algorithm, Pebble, and Breadcrumb based on the Twitter, and DBLP dataset. We further use the TPC-H dataset to judge the explanation quality of Breadcrumb's explanations, since it is the only dataset that comes with predefined queries. By injecting errors into the queries, we can assess Breadcrumb's explanation quality.

8.2 Tree-pattern matching

We start the experimental evaluation with the tree-pattern matching algorithm. This section is based on [DH20a]. We conduct the evaluation on cluster 2. Furthermore, we briefly describe the evaluation workload before we present evaluation results. We leverage the the evaluation workload to conduct five experiments that validate the following five claims. First, the algorithm’s schema matching phase is independent of the data size. Consequently, the runtime of the schema-matching phase does not increase with increasing data size. Second, the algorithm’s overall runtime scales with the input dataset size. Third, the algorithm’s overall runtime scales with the provided compute resources. Fourth, the algorithm has competitive runtime to Spark native queries that compute the same result. Finally, the tree-patterns reduce the effort needed to write big data query pipelines with and without the tree-pattern matching algorithm. Next, we briefly describe the evaluation workload.

8.2.1 Workload

We define seven evaluation queries based on the Twitter and DBLP datasets and briefly summarize them in Table 8.2. For each scenario, we define a tree-pattern and apply it to the test data. Our published work [DH20a] provides further details on the scenarios and a detailed scenario description including the tree-pattern definition is available online¹. The seven queries are split into five Twitter scenarios (T1 to T5) and two DBLP scenarios (D1 and D2). The table’s leftmost column provides a short, informal description of each scenario. The central column holds the number of the schema matches.

For instance, in scenario T1, the tree-pattern query requests all distinct names occurring in the Twitter datasets. Data engineers may leverage this query to explore the dataset. In addition to the root node, the query’s tree-pattern has just one *name* node. These nodes are connected with an

¹https://www.ipvs.uni-stuttgart.de/departments/de/resources/pebble/pebble_tpm_workload.pdf

S	$ \mathcal{M} $	Description (detailed descriptions available in [DH20a])
T1	14	Returns all distinct <i>names</i> occurring in the input
T2	10	Retrieves all <i>names</i> of persons who either have authored a user-mentioning tweet or are themselves mentioned in at least one non-retweeted tweet
T3	1	Computes the <i>user</i> and <i>retweet_count</i> of tweets with at least three <i>hashtags</i> longer than five characters
T4	1	Computes a nested list of <i>hashtags</i> for each <i>user_mentions</i>
T5	1	Selects tweets with at least two <i>user_mentions</i> that have <i>name</i> longer than 5 characters and at least two <i>hashtags</i> containing “BTS”
D1	4	Selects all inproceedings with their proceedings from the last century that have more than 10 citations
D2	1	Selects all authors that have at least 2 aliases containing “Mill”

Table 8.2: Evaluation scenarios for the tree-pattern matching algorithm

ancestor-descendant relationship. This pattern matches 14 times on the input dataset because the attribute *name* appears 14 times in the input dataset. Once the tree-pattern matching algorithm has computed all non-distinct names, the query leverages Spark’s *distinct* transformation to obtain the distinct names.

While T1 has a simple tree-pattern, other scenarios, such as T2 or T3 have more complex patterns with multiple nodes and constraints defined on the nodes. In summary, we have created the scenarios in such a way that they contain simple and complex tree-patterns with and without value and cardinality constraints so that all our tree-pattern features occur in the workload. We leverage the scenarios to conduct the following five experiments.

8.2.2 Schema-matching runtime

In the first experiment, we show that the runtime of the schema-matching phase is independent of the dataset size. Figure 8.1 shows the respective runtime (y-axis) with increasing dataset sizes (x-axis). Across the scenarios, it is roughly constant over all dataset sizes, as it is computed independently from the data. Furthermore, it is negligible (< 50 ms) compared to the overall scenario execution time, which is multiple seconds to a few minutes.

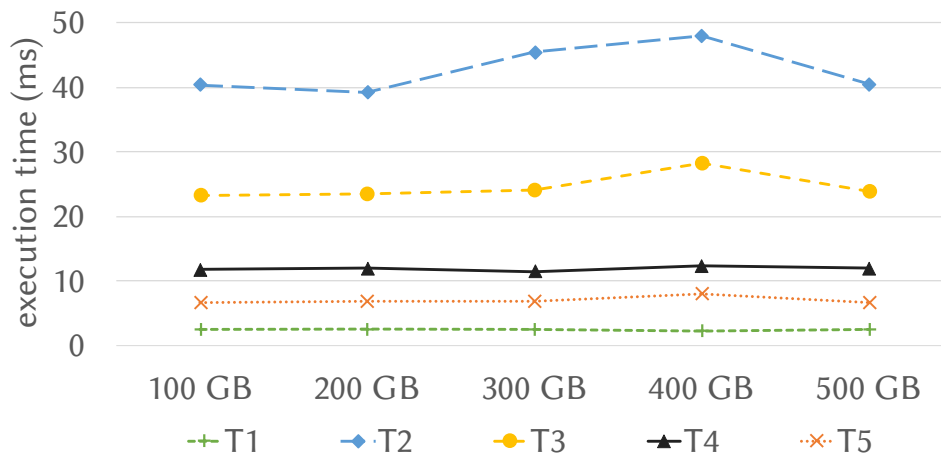


Figure 8.1: Runtime of the schema-matching with increasing dataset sizes

The matching phase is the shortest with the simplest tree-pattern and increases with the complexity of the tree-pattern and the number of potential schema matches. In scenario T1, the algorithm has to match only the name node on the schema. It does not need to check dependencies between nodes in the pattern. Thus, the runtime is only about 2.5 ms, even though the pattern matches 14 times. In contrast, in scenario T2, the algorithm has to check multiple paths from the tree-pattern leaves to the root. Further, it has to validate at least 10 schema matches, so that the runtime for T2 is up to 50 ms. The runtimes for scenarios T3-T5 range between the runtimes of T1 and T2, even though their tree-pattern is more complex than the one of T2. However, compared to T2, the algorithm has to validate fewer schema matches on the input schema. The scenarios D1 and D2 are not displayed, but their schema-matching runtime was constantly below 5 ms since the DBLP schema has significantly fewer attributes than the Twitter schema.

In conclusion, the conducted experiment confirms that the schema matching phase is independent of the data size and the size of the schema always remains the same regardless of the data size. However, the runtime varies significantly across the scenarios due to different tree-pattern complexity and schema complexity. Nonetheless, even for large schemata, such as the Twitter schema, the schema-matching runtime is negligible compared to the data-matching runtime, as we show next.

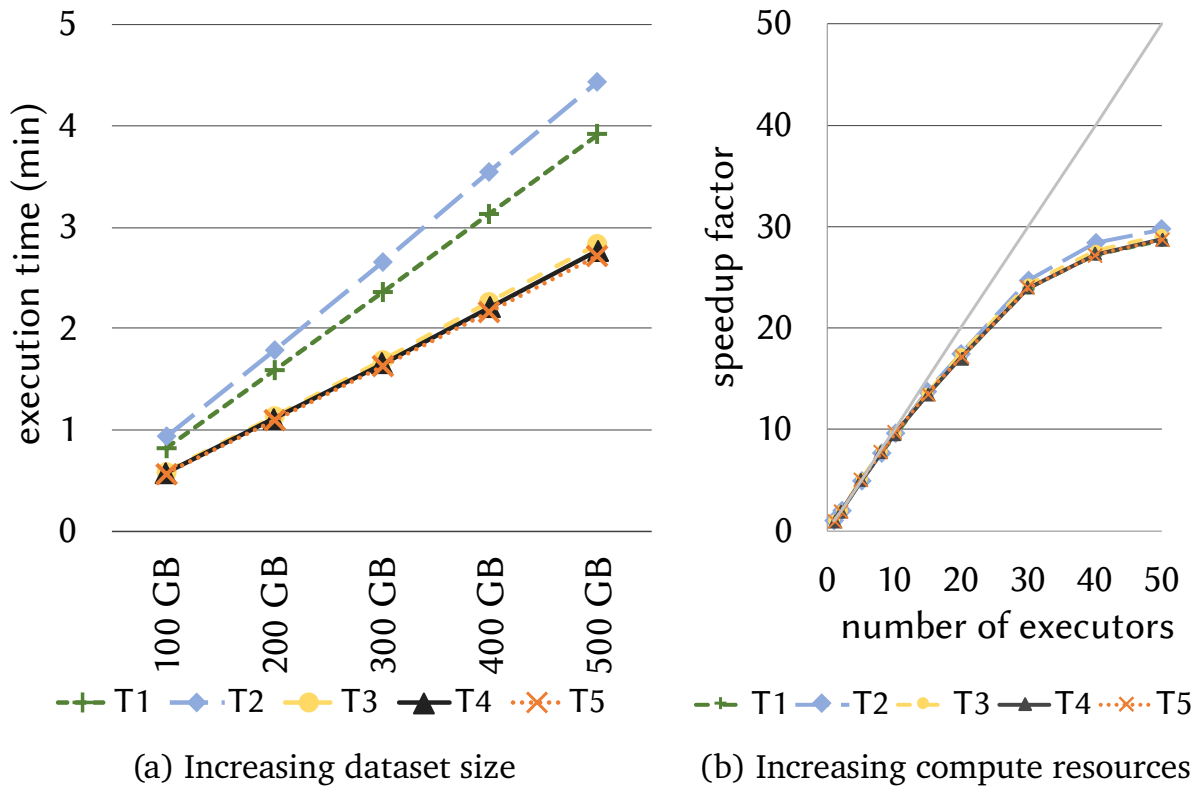


Figure 8.2: Runtime of the tree-pattern matching algorithm [DH20a]

8.2.3 Scalability with increasing dataset size

In the second experiment, we measure the overall runtime of the tree-pattern matching scenarios, while we gradually increase the input data from 100GB to 500GB. The results for the Twitter datasets are shown in Figure 8.2a. We observe that our tree-pattern matching implementation scales linearly with the input data size across all Twitter scenarios. The DBLP scenarios D1 and D2 that are not displayed also scale linearly. The scenarios T3, T4, and T5 have similar runtimes because they have just one match on the schema. As a consequence, the data matching boils down to simple attribute accesses in the processed data. Scenario T2 has the highest runtime because the algorithm validates ten schema matches $M \in \mathcal{M}$ on the input data. For each match, the tree-pattern matching algorithm has to check all nodes in the pattern with all their constraints. That is more computationally intensive than validating 14 matches on a single attribute (T1) and than checking

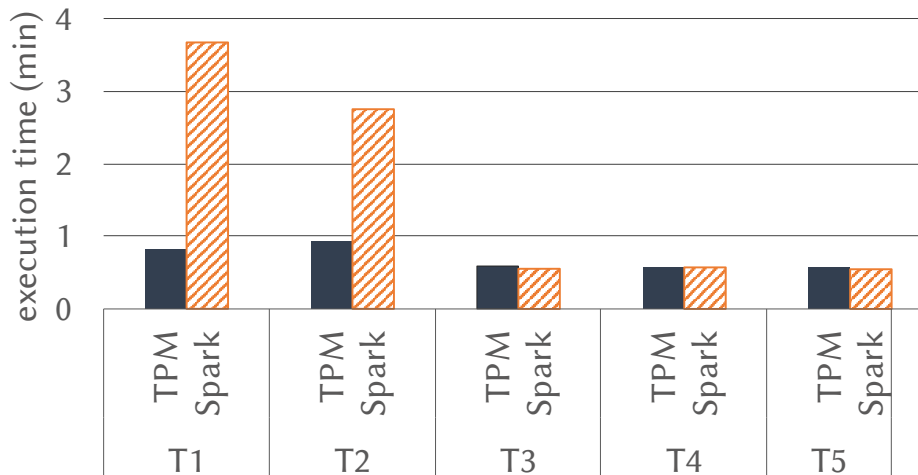


Figure 8.3: Runtime with tree-pattern matching (TPM) and without (Spark) [DH20a]

only one schema match (T3-T5). Hence, we have identified three properties that directly impact the runtime: the dataset size, the complexity of the tree-pattern, and the number of schema matches. This experiment further confirms that our algorithm scales linearly with increasing dataset sizes.

8.2.4 Scalability with increasing compute resources

In the third experiment, we keep the dataset size constant at 100GB and increase the number of executors to study horizontal scaling on the compute resources. We report the speedup results for the Twitter scenarios in Figure 8.2b. The results for the DBLP scenarios are not displayed but in accord with the ones of the Twitter scenarios. The speedup s is $s = \frac{r_1}{r_n}$, where r_1 is the scenario's runtime with one executor and r_n is the runtime with n executors. The ideal speedup for n executors is n itself. We indicate it with the diagonal grey line in Figure 8.2b. All scenarios have ideal speedup up until $n \approx 15$. Then, the speedup degrades, mainly because disk-I/O is dominating the runtime and the executors share 12 dedicated SSD disks. In conclusion, this experiment shows that the tree-pattern matching algorithm scales linearly with increasing compute resources assuming that the compute resources include all compute resources, not just the CPU and main memory.

8.2.5 Runtime comparison with plain Spark

The fourth experiment compares the runtime of the pipelines with tree-pattern matching (TPM) and with standard Spark operators (Spark). Fig. 8.3 shows the results for the Twitter scenarios. In scenarios T1 and T2, TPM is two to four times faster than Spark because the Spark pipelines unify intermediate results in a final step. More precisely, they trigger data shuffling across the compute nodes, which imposes unnecessary overhead. The TPM approach does not shuffle the data.

Since scenarios T3 to T5 only yield a single schema match, Spark does not need to merge any intermediate results in these scenarios. As a consequence, no additional shuffling occurs and TPM is marginally slower than plain Spark. This performance gap grows in the DLBP scenarios. In D1 and D2, Spark is about 30% faster than TPM. In these scenarios, the Spark pipelines are highly optimized, benefitting from, e.g., filter push-downs and custom code-generation for each pipeline. For instance, in D1 and D2, the filters on year are applied before joining the proceedings and inproceedings in plain Spark. In contrast, in the TPM pipelines, this push-down is not applied because the filter condition is encoded in the tree-pattern. Overall, our tree-pattern matching algorithm has competitive runtimes, even though it does not benefit from the described optimizations.

8.2.6 Query pipeline complexity

In addition to the preceding experiments, we conduct an experiment to assess the effort needed to define query pipelines. We compare the number of operators in the pipeline. Table 8.3 shows the numbers with and without tree-pattern matching in columns *TPM Ops* and *Spark Ops*, respectively. They do not include read and write operations at the beginning and end of each pipeline.

With the tree-pattern matching, the pipelines typically have the single tree-pattern matching operator. In scenarios T1 and D1, they have two operators in total since a deduplication operator succeeds the tree-pattern matching

Scenario	TPM Ops	Spark Ops
T1	1	33
T2	2	29
T3	1	6
T4	1	6
T5	1	12
D1	2	8
D2	1	5

Table 8.3: Number of operators in the query pipeline, with and without tree-pattern matching [DH20a]

operator. Without the tree-pattern matching, they have 5 to 33 operators. The pipelines in T1 and T2 consist of 33 and 29 operators, respectively. These numbers are particularly high since the pipelines consist of one projection for each occurrence of the attribute *name* in the input data. Furthermore, the pipelines have to flatten *name* attributes in nested collections. In D1, the number is high since the *year* and the *citation* attributes occur multiple times in the DBLP schema and each occurrence needs a separate operator. Leveraging tree-patterns in these scenarios is particularly beneficial since the node that holds an attribute’s name appears just once in the tree-pattern. When developers use plain Spark, they must conduct multiple manual steps. First, they have to find all occurrences of an attribute in the schema, then they have to write sub-pipelines for each attribute. Eventually, they have to integrate these pipelines. This process is error-prone and time-consuming.

In scenarios T3-T5 and D2, the number of *Spark Ops* is higher than the *TPM Ops* because the pipelines need to express cardinality constraints and value constraints at the same time. More precisely, the tree-patterns in these scenarios define a cardinality constraint over at least one nested collection whose elements are further constrained on a certain value. The Spark pipelines have to implement these constraints. Therefore, they flatten the nested collection, filter on the attributes with the value constraints, and aggregate the filtered values to enforce the cardinality constraints. Defining this pipeline is once again more complex than defining a tree-pattern.

In summary, in all the above scenarios, tree-matching significantly reduces the complexity of pipelines, allowing for a faster pipeline development process, lower pipeline maintenance effort, and higher overall productivity. We further leverage the tree-pattern matching algorithm in Pebble and Breadcrumb whose evaluation results we discuss next.

8.2.7 Discussion

In this section, we have validated that (i) the runtime of the schema matching phase is independent of the dataset size. This is a key property of our algorithm and a prerequisite to scalability on large datasets. (ii) Furthermore, the tree-pattern matching scales linearly with the data size and (iii) scales linearly with increasing compute resources. Therefore, our algorithm is the first to scale with increasing dataset sizes on big data analytics clusters. (iv) Fourth, the algorithm yields better or comparable execution times than plain Spark as long as Spark does not benefit from built-in optimizations. (v) Finally, even in the cases, in which Spark currently outperforms our tree-pattern matching implementation, it is appealing to use our algorithm because it likely yields a simpler query pipeline.

8.3 Explanations for existing data

We conduct multiple experiments to validate our claims that Pebble scales to large datasets and provides more comprehensive explanations than other existing solutions. The evaluation is based on [DH20b] and consists of five experiments conducted on cluster 1. The first two experiments measure Pebble's runtime and space overhead during provenance capture to validate Pebble's scalability properties during provenance capture. The third experiment measures the time to query explanations based on the previously captured provenance annotations. It shows that Pebble efficiently computes the explanations from the captured provenance. The fourth experiment compares Pebble's capture overhead with Titian's overhead in a microbenchmark [IST+15] to show that Pebble imposes comparable overhead even

S	Description (detailed descriptions available in [DH20b])
T1	filters tweets containing the text <i>good</i> , flattens and groups by the mentioned users to collect a bag of complex tweet objects
T2	flattens the nested lists: hashtags, media, and user mentions
T3	yields for each users a list tweeted texts that they have authored or are mentioned in
T4	associates all occurring hashtags with the authoring and mentioned users
T5	finds all users that tweet about <i>BTS</i> , and are mentioned in a <i>BTS</i> 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 8.4: Evaluation scenarios for the Pebble algorithm

though it computes more precise explanations for nested datasets. In the final experiment we leverage Pebble to show that Pebble enables two real-world use-cases, in which existing provenance-based solutions fail to provide sufficiently comprehensive explanations.

8.3.1 Workload

Before we start the evaluation, we introduce the query scenarios used in the following experiments. As shown in Table 8.4, we define ten scenarios. The scenarios T1 to T5 are defined on the Twitter dataset and D1 to D5 on the DBLP data. The table provides a short description of each scenario and points to [DGHL19] for more details. Furthermore, the scenario descriptions are available online¹. The scenarios are designed in such a way that they cover all supported operators and different combinations of them.

For instance, the query pipeline in scenario T3 computes a relation of users appearing in the Twitter dataset. These users occur in at least one tweet that was retweeted. They have either authored the tweet or are mentioned in the

¹https://www.ipvs.uni-stuttgart.de/departments/de/resources/pebble/pebble_edbt_workload.pdf

tweet. For each of these users, the pipeline computes a nested relation that holds the texts of the tweets, in which they occur. This scenario contains a filter, projection, union, tuple flatten, relation flatten, tuple nesting, and relation nesting operator. It queries an explanation for a duplicate tweeted text in a single user’s nested relation. Therefore, it conducts a very selective provenance query.

While T3 has a complex pipeline and highly selective provenance query, other scenarios exactly the opposite properties. T1 and T2 have rather simple query pipelines that mainly consist of filters, projections, and relation flatten operators. While T1 has a non-selective provenance query, T2’s provenance query is very selective. We describe more scenarios during the evaluation to explain the measured results. We also refer the reader to [DGHL19] for more detailed workload descriptions. We leverage all ten scenarios for the following quantitative evaluation.

8.3.2 Runtime overhead and scalability

The first experiment studies Pebble’s runtime overhead imposed by the lightweight provenance capture on the defined scenarios. The experiment’s goals are twofold. It shows (i) that Pebble scales over the input data size and (ii) that Pebble keeps the relative runtime overhead constant with increasing data size.

To show Pebble’s scalability, we report results for input datasets of up to 500GB. Related solutions that yield explanations for nested data [ADD+11; ZAI19] have only been evaluated on datasets of a few hundred Megabytes to a few Gigabytes. Hence, we report results on input data that is more than 100 times larger than the data used to evaluate the related solutions.

We measure the execution time for each scenario with and without provenance capture on datasets from 100GB to 500GB and report the results on the Twitter workload in Figure 8.4 and the DBLP workload in Figure 8.5. The solid dark grey part of each bar shows the runtime that Spark requires without provenance collection. The textured light grey part on top of each solid bar shows Pebble’s provenance capture overhead. The percentages on

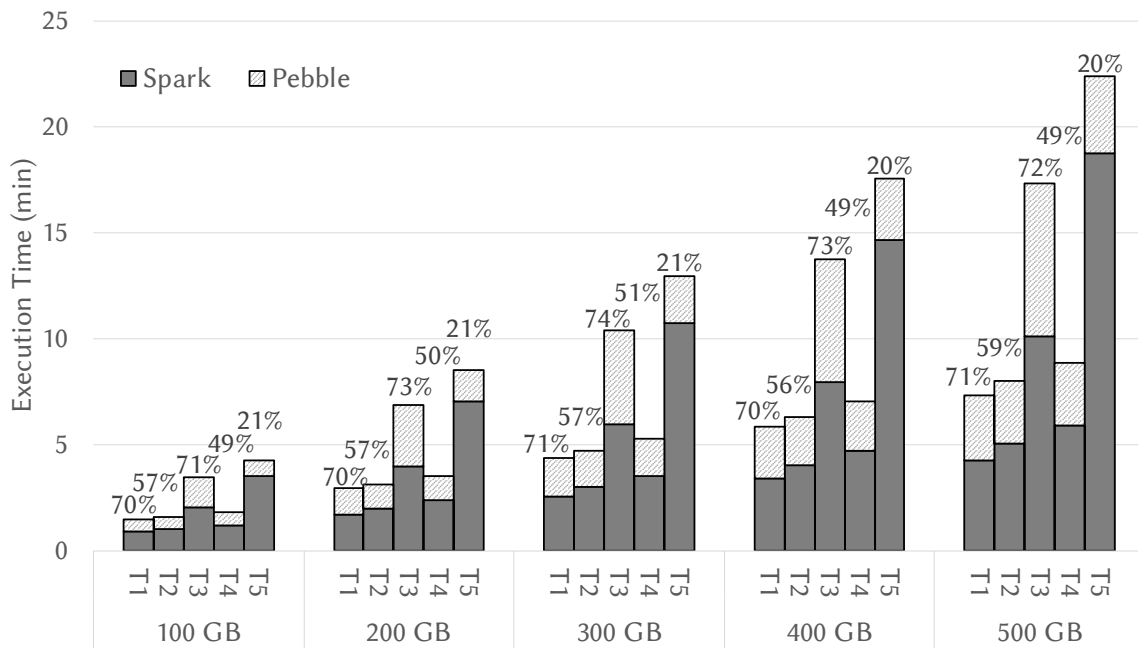


Figure 8.4: Runtime overhead on Twitter dataset [DH20b]

top of the textured bars indicate the relative overhead between the former and the latter series of measurements.

The runtime peaks at 22 minutes for the Twitter dataset and 32 minutes for the DBLP dataset. Constantly across all scenarios, the runtime increases when Pebble captures the provenance annotations, since the capturing imposes additional workload. The execution time with and without provenance grows linearly with the data size. The overhead percentages indicate that the relative capture overhead is quite constant with increasing data sizes.

However, we observe that the relative overhead varies significantly among the scenarios. It ranges from 8% (D3) up to 75% (T3). In scenario T3, the overhead is particularly high because T3 reads the input tweets twice to perform a union operation. As a consequence, Pebble annotates the input data twice during provenance capture. That prevents Spark from optimizing the query pipeline so that it reads the input data just once rather than twice. In scenario D3, the overhead is particularly low since I/O operations mask the provenance capture overhead. In this scenario, Spark spills large final and intermediate results to disk. Compared to the disk I/O, the time to compute the extra provenance is small.

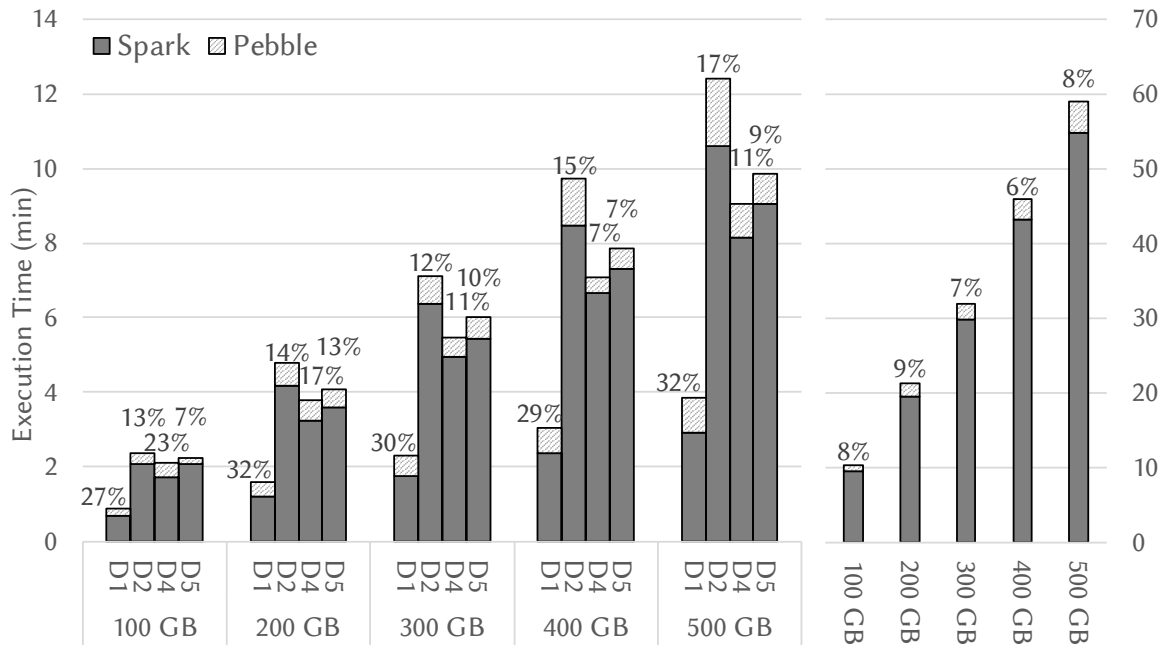


Figure 8.5: Runtime overhead on DBLP dataset (right: D3) [DH20b]

To generalize our runtime overhead observations in the mentioned scenarios, we have conducted a few micro-benchmarks on individual operators. They reinforce the following general observations. Recall that by definition, the filter, select, union, join, and flatten have constant provenance annotation overhead since they compute at most two annotation columns per processed top-level tuple. For these operators, 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. Unlike the mentioned operators, the aggregation and relation nesting operators have varying annotation overhead. It depends on the number of input tuples that are reduced to a single output tuple. For these operators, Pebble stores a nested relation with all tuple identifiers that contribute to the output tuple. This collection typically is orders of magnitude larger than the output tuple without annotation would be. Consequently, the observed runtime overhead exceeds 100% of the actual execution time for the aggregation and relation nesting. However, even this overhead is negligible when disk I/O operations dominate the execution time.

We conclude that Pebble (i) scales with input data sizes far beyond those that other existing solutions have supported and (ii) keeps the relative overhead constant across different input data sizes. Furthermore, the relative runtime overhead depends on the operators in the query and the amount of disk I/O needed to process the query.

8.3.3 Space overhead

The second series of measurements captures the space overhead that Pebble imposes on Spark to store the 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-based solutions that only trace provenance at the granularity of top-level tuples.

We measure Pebble’s space overhead for each scenario on the 100GB real-world datasets. We further compare Pebble’s lineage overhead with Pebble’s structural provenance overhead. The lineage overhead is the overhead that lineage solutions, such as Titian [IST+15], Newt [LDY13], or RAMP [IPW11] would compute when applied to the scenarios. Pebble’s structural provenance additionally holds positional information of nested elements and the access and manipulation paths on the schema level. We write all dataframes containing intermediate tuple identifiers onto disk and store them in Apache Parquet files. We sum the sizes of each of the spilled dataframes to obtain the total size of the produced structural provenance.

The measured results for both datasets are shown in Figure 8.6. The dark grey part of each bar shows the size of the lineage for top-level items. 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 two graphs have different scales since Pebble mainly associates identifiers to top-level tuples. The top-level tuples in the Twitter dataset have about 1000 attributes, whereas the items in the DBLP dataset have less than 50

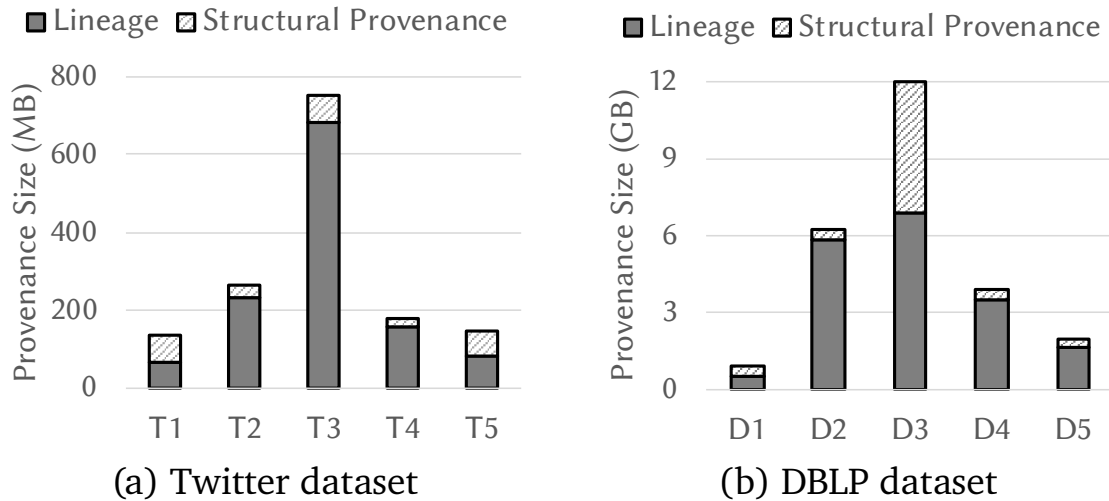


Figure 8.6: Size of collected provenance [DH20b]

attributes. Thus, 100GB of DBLP data contains more than 100 times as many top-level tuples as 100GB of Twitter data. Therefore, Pebble stores more than 100 times more annotations for the DBLP scenarios than the Twitter scenarios. Hence, the DBLP provenance is orders of magnitude larger than the Twitter provenance. We conclude that the provenance size significantly depends on the number of tracked top-level data items in the input.

Furthermore, the provenance sizes significantly differ among the scenarios of the same dataset. For instance, the provenance’s size in scenario T3 amounts to 750MB. This is 5.5 times the size of T1’s provenance. There are three reasons for the different size: (i) As mentioned above, in T3, Pebble annotates the input data twice; (ii) In T3 the pipeline has 7 processing steps for each of which Pebble collects annotations, whereas T1’s pipeline consists of only 5 steps; (iii) The filter in T1 reduces the total amount of tracked data items early in the pipeline. Therefore we conclude that space overhead also depends on the number of operators in a program and the number of top-level items in intermediate results.

Next, we compare the runtime overhead with space overhead. As shown in Figure 8.4, scenarios T3 and T1 have comparable runtime overheads of around 70 - 75%. They are the highest across all scenarios. While scenario T3 also yields the largest provenance space overhead, T1 has a five times

smaller provenance overhead. Similarly, the relationship of the runtime and space overhead is the inverse for scenarios D3 and D1. D3 has the largest and D1 the smallest provenance space overhead. 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.

We finally compare the space overhead of lineage of top-level tuples with the one of structural provenance. In all scenarios but D3, structural provenance takes less than 200MB additional space, even in scenarios where lineage itself takes Gigabytes. D3 is an exception since a relation flatten operator appears early in the pipeline. While processing this operator, Pebble captures not only the identifiers of the top-level tuples but also the identifiers (i.e., positions) of the nested tuples that are flattened. Additionally, a highly selective join operator succeeds the flatten operator in D3's pipeline. It significantly reduces the number of output tuples and, thus, causes the comparatively high size difference between lineage and structural provenance.

We draw the following generalized conclusions from the above experiments. (i) The provenance size depends on the number of top-level tuples in the input and intermediate results. (ii) Furthermore, the provenance's space overhead may not correlate with its 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. (iii) Additionally, the size difference between lineage and structural provenance is small in many practical scenarios. However, it can noticeably increase when flatten operators store positions of nested tuples that lineage solutions do not capture. All in all, Pebble's structural provenance overhead is comparable to overhead of lineage-based solutions for big data analytics systems. However, Pebble can derive explanations at the granularity of individual nested attributes rather than mere top-level tuples.

8.3.4 Explanation query time

The third experiment assesses the execution time that is needed to query explanations from the captured provenance on the 100GB datasets. To

show that Pebble’s holistic approach is faster than a fully lazy approach, we compare two approaches to query the explanations. We report runtime results on Pebble’s holistic approach and on a lazy provenance query approach. Pebble’s holistic approach eagerly captures lightweight provenance during pipeline execution and builds detailed explanations once the explanations are queried. Pebble also implements a fully lazy explanation query approach. It completely computes the annotations and explanations once the explanations are requested. One can consider this approach an extension of PROVision [ZAI19] to our processing pipelines.

We show the explanation query time for both approaches in Figure 8.7. Figure 8.7(a) shows the Twitter scenarios and Figure 8.7(b) shows the DBLP scenarios on the x-axis. The y-axis reflects the explanation query time. The reported times include the tree-pattern matching on the pipeline result and the backtracing of the structural provenance. The dark bars are labeled with *eager* and represent the times from our holistic approach. The brighter bars show the times for the *lazy* approach.

Figure 8.7 does not show dedicated times for each of the two explanation query steps because 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, we cannot reliably time each step separately. However, we refer to the experiment in Section 8.2 to get an impression of the tree-pattern matching performance.

Constantly across all scenarios, querying structural provenance eagerly takes more time than the actual program execution (cf. Figure 8.4 and Figure 8.5). We identify two reasons for this behavior: (i) the backtracing procedure performs a join operation for each operator in the actual program. That happens even for computationally less expensive operators such as filters and selects. (ii) Backtracing has to manipulate the provenance trees for each operator.

When we compare the performance of Pebble’s holistic capture and query approach with the completely lazy query approach, Pebble’s holistic explanation querying approach is consistently faster than the lazy approach. In scenarios T3, T5, and D3 the difference ranges between a factor of four

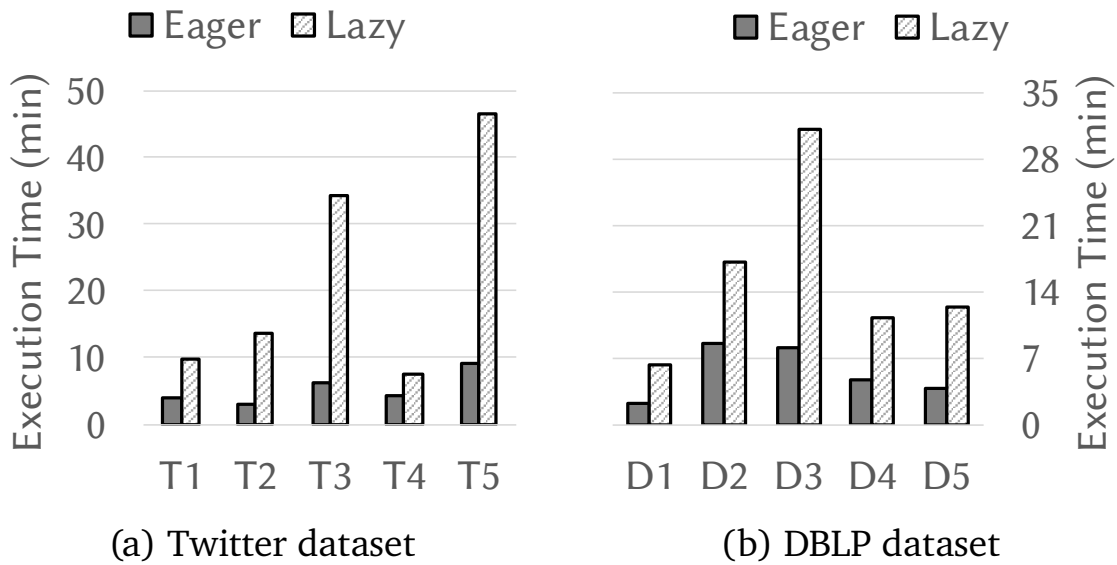


Figure 8.7: Runtime of Pebble’s backtracing [DH20b]

to seven. That has 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 adds 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 series of measurements we draw the following conclusions. Pebble’s holistic explanation querying approach outperforms the lazy querying approach. Furthermore, lazily querying explanations becomes less attractive with increasing pipeline complexity. It is less time-consuming to rerun a scenario from scratch with provenance capture and query the explanation eagerly than to lazily query the explanations.

8.3.5 Comparison with Titian

In a micro-benchmark, we compare Pebble with Titian [IST+15] regarding the provenance capture overhead. We choose Titian since it is the only other provenance solution that is fully integrated into a big data analytics system, such as Spark. We find it hard to conduct a more comprehensive

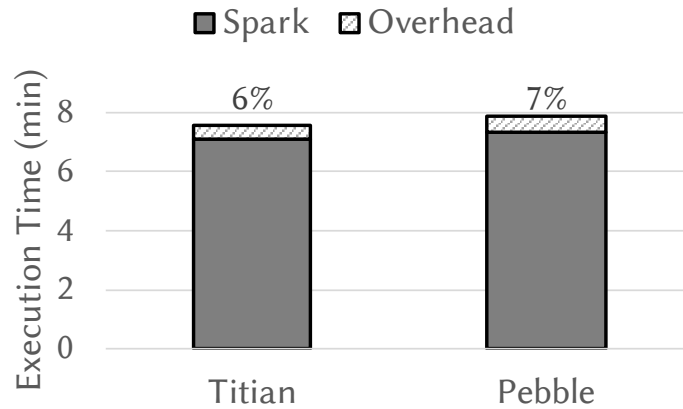


Figure 8.8: Runtime comparison Titian [DH20b]

and reliable comparison because 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 dataframes offered by the SparkSQL API.

Figure 8.8 shows the execution results. Without provenance computation, the query pipelines have an average runtime of 7.13 seconds on RDDs and 7.36 seconds on dataframes. The overall runtime 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 has marginally higher runtime overhead than Titian, even though it is capturing structural provenance. The additional percentage point resides in the different implementation approaches. Yet, the overhead for provenance collection on flat data is comparable for both solutions. However, Pebble outperforms Titian in other aspects. It additionally supports nested data and structural provenance at the granularity of attributes rather than tuples. These features play a key role in the following use-cases.

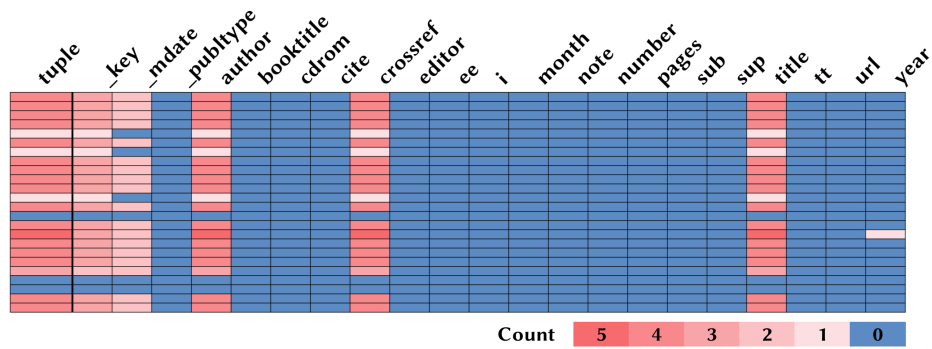


Figure 8.9: HeatMap for 25 tuples in the DBLP inproceedings dataset after running scenarios D1-D5 [DH20b]

8.3.6 Use-cases for structural provenance

To assess the practical benefits of structural provenance, we illustrate two real-world use-cases, in which Pebble outperforms other existing provenance solutions because it provides more comprehensive explanation with a finer granularity. We prototypically implement a data-usage and an auditing use-case to analyze how they benefit from structural provenance. We start with the data-usage case.

Data-usage. In the first use-case, our goal is to find frequently accessed and manipulated data to optimize the data layout on disk. Based on these optimizations queries run faster in the future. We do not only want to find out, which data is frequently accessed but also which attribute combinations are accessed together to derive data-usage patterns. Further, we like to identify hot and cold data. Hot data is frequently accessed, whereas cold data is barely or never accessed by a query workload. Pebble can identify the data-usage patterns and highlight hot and cold data. To obtain the patterns and the hot and cold data, we run the DBLP scenarios D1 through D5 and aggregate the explanations of the scenarios.

Figure 8.9 shows a heatmap of 25 randomly selected tuples from the DBLP inproceedings dataset after running test scenarios. The first column represents the access to the entire tuple. The second to last columns represent the access to individual attributes within the tuple. The heatmap only shows

top-level attributes due to space constraints. The more often the query workload accesses an attribute, the deeper becomes the red color in the cells of Figure 8.9. If an attribute is not accessed at all, it is colored blue.

All but three top-level items have influenced at least one query result. However, only a fraction of all attributes contributes to the results. The *_key*, *_mdate*, *author*, *title*, and *year* attributes are the only accessed attributes. Therefore, in this example, a horizontal (tuple-based) partitioning of hot and cold tuples may not significantly improve system performance or save space in memory. In contrast, a vertical (column-based) partitioning of hot and cold attributes is likely to improve system performance significantly. Furthermore, 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 likely benefits from storing these items next to each other.

A comparison with other provenance solutions highlights the perks of Pebble. Lineage-based provenance solutions such as Titian [IST+15] only provide the tuple counters. These tuple counters may even over-estimate the tuple access when applied to nested data. PROVision [ZAI19] also provides the tuple counters. However, they are not over-estimated. Unlike the previously mentioned solutions, Lipstick [ADD+11] can identify attribute counters. However, Lipstick does not reveal information on access and manipulation and, thus, misses influencing attributes such as the attribute exclusively accessed in filter conditions. Pebble correctly identifies the attribute counters and takes influencing attributes into account. Hence, it is the first provenance-based solution that faithfully captures data-usage patterns on nested input data.

Auditing. We look into auditing to illustrate Pebble’s strengths in a second use-case. In brief, auditing has two purposes. It finds leaked data and the person who has leaked the data. Pebble addresses the first purpose. It identifies directly or indirectly leaked data. Directly leaked data is the data whose values got exposed. Indirectly leaked data is the data whose values can be inferred, e.g., by combining the leaked data with additional data from other sources. This process is called a reconstruction attack.

Let us assume the DBLP scenarios were designed to obtain sensitive personal information on researchers. Then, all data in Figure 8.9 are leaked whose counts are 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* value whose count is one. It is marked as influencing since it does not contribute to any result item in D1 to D5. However, knowing that the *year* is accessed helps to assess the risk of reconstruction attacks.

In the context of this use-case we compare Pebble with other existing provenance solutions. Lineage solutions and PROVision [ZAI19] 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 [ADD+11] potentially misses leaked information since it misses influencing attributes like the *year*. Thus, neither of the mentioned solutions allows for such a fine-grained risk assessment as Pebble does.

We conclude, that Pebble enables novel use-cases because its explanation quality outperforms the quality of other solutions in two aspects. First, it provides explanations at the granularity of individual nested attributes rather than mere tuples. Second, by implementing structural provenance, it distinguishes between access and manipulation of data.

8.3.7 Discussion

In conclusion, the experiments explained in this section validate that Pebble scales to large datasets in big data analytics systems. Furthermore, it provides more comprehensive explanations than other existing solutions, which enables novel use-cases. More precisely, we have shown that Pebble scales with increasing input data size because the relative runtime overhead stays constant. We have further shown that the provenance's space overhead may not correlate with the runtime overhead since other factors, such as disk I/O mask the runtime overhead for provenance capture. Furthermore, Pebble's holistic approach to capturing the provenance and query the expla-

nations outperforms fully lazy provenance solutions. Even though Pebble provides explanations that are more comprehensive than others, it imposes only marginally higher runtime overhead than Titian, which only collects provenance for flat tuples. Pebble further supports nested data and structural provenance. It provides explanations at the granularity of attributes rather than tuples. Hence, it enables novel use-cases such as finding data usage patterns and auditing.

8.4 Explanations for missing data

In this section, we assess the key properties of the Breadcrumb algorithm leveraging cluster 2. The results presented in this section are based on our publications [DLHG21a; DLHG21b]. We describe three experiments, here. First, we conduct an experiment that shows Breadcrumb’s scalability on increasing dataset sizes. Second, we assess the runtime impact of schema alternatives. For that purpose, we conduct two series of measurements. The first series sheds light on the runtime impact with schema alternatives switched on and off. The second series of measurements gradually increases the number of alternatives to show scalability with an increasing number of schema alternatives. Third, we discuss the quality of the provided explanations. We compare Breadcrumbs explanations computed with and without schema alternatives with an improved version of the WhyNot algorithm [CJ09]. All experiments are conducted on the nested datasets described above. We refer the interested reader to [DLHG21b] for further experiments on flat relational data.

In the following evaluations, we study the explanations that Breadcrumb yields with (RP) and without (RPnoSA) multiple schema alternatives. We further compare these to the explanations with WN++ in the quantitative evaluation. WN++ extends the lineage-based Why-Not algorithm [CJ09] in two aspects. It scales to the dataset sizes described in this work and it supports nested data. Given the scenarios and the algorithms, we start the quantitative evaluation next.

S	SAs	Description (detailed descriptions available in [DLHG21b])
D1	2	All authors and titles of papers that are published at SIGMOD
D2	2	Number of articles for authors who do not have <i>Dey</i> in their name
D3	2	Lists all author-paper-pairs per booktitle and year
D4	2	Papers per author that have published through ACM after 2010
D5	2	List of (homepage) urls for each author
T1	2	List of tweets providing media urls about a basketball player
T2	2	All users who tweeted about <i>BTS</i> in the US
T3	2	Hashtags and medias for users that are mentioned in other tweets
T4	2	Nested list of countries for each hashtag of tweets containing <i>UEFA</i>
T _{ASD}	4	ASD example [SAC+17]: flatten, filter, project quoted tweets (2 mods)
Q1	6	TPC-H query 1 with one modified aggregation
Q3	12	TPC-H query 3 with two modified selections
Q4	12	TPC-H query 4 with a modified selection and aggregation
Q6	6	TPC-H query 6 with one modified selection
Q10	2	TPC-H query 10 with two modified selections and a modified projection
Q13	1	TPC-H query 13 with one modified join

Table 8.5: Evaluation scenarios for the Breadcrumb algorithm

8.4.1 Workload

Before we present the evaluation results, we briefly introduce the evaluation scenarios. Table 8.5 summarizes the overall sixteen scenarios. The leftmost column holds the scenario label, the center column the number of schema alternatives including the original schema, and the rightmost column provides a brief description of the scenario. We define five DBLP scenarios D1-D5 and four Twitter scenarios T1-T4, which cover a wide range of supported operators and their combinations. For each of these scenarios, we define one schema alternative in addition to the original schema alternative so that they have two alternatives in total.

We derive the remaining scenarios T_{ASD}, Q1, Q3, Q4, Q6, Q10, and Q13 from existing scenarios. T_{ASD} is derived from the running example on adaptive schema databases [SAC+17]. The scenarios starting with a Q are derived from the respective TPC-H queries [Tra09] over the nested TPC-H dataset [PCW17]. We introduce errors to these scenarios and use the unmodified queries as gold standard. We assess the quality of the explanations based on this gold standard. Next, we describe the T_{ASD} and the TPC-H workload, particularly Q3 and Q10 in more detail.

Scenario T_{ASD} originates in the work [SAC+17] on adaptive schema databases. An adaptive schema database (ASD) extracts and refines relational schemata from semi-structured or unstructured data. It computes a set of possible relational candidate schemas for the input data and leverages provenance to identify the ambiguity on the existing output data. T_{ASD} is based on the Twitter dataset and extracts one relation each for the nested retweeted tweets and the nested quoted tweets. To extract the retweeted tweets the ASD (i) flattens them, (ii) filters a non-null retweet count, and (iii) projects only the attributes from the retweet. As indicated in Table 8.5, we add two errors to T_{ASD} to reflect the ambiguity between retweets and quotes. We flatten the quoted tweets instead of the retweeted tweets and filter on the quote count instead of the retweet count. We further provide two attribute alternatives. The retweeted tweet is an alternative to the quoted tweet and the retweet count is an alternative to the quote count. Breadcrumb derives four schema alternatives from the two attribute alternatives. These schema alternatives are important to systematically find a certain tweet that we ask for in T_{ASD} 's why-not question. It is initially absent from the T_{ASD} 's result.

We derive the scenarios Q1, Q3, Q4, Q6, Q10, and Q13 from the TPC-H queries with the same names [Tra09] by removing the top-k and sorting operators from the queries. We further introduce errors into these queries as mentioned in Table 8.5, so that Breadcrumb has to find them. For that purpose, Breadcrumb may leverage schema alternatives. Breadcrumb computes the schema alternatives from the following three sets of attribute alternatives:

- $\{l_discount, l_tax\}$,
- $\{l_shipdate, l_commitdate, l_receiptdate\}$, and
- $\{o_orderpriority, o_shippriority\}$.

As shown in Table 8.5, Breadcrumb computes up to 12 schema alternatives in the mentioned scenarios, depending on the attributes occurring in the scenario. For instance, Q4 references attributes from all three sets. Thus, it

has 12 schema alternatives. In contrast, Q13 references neither attribute. Hence, it only has the single original schema alternative.

Scenarios Q3 and Q10 become particularly important during the qualitative evaluation. Hence, we describe them here in a bit more detail. Scenario Q3 computes unshipped orders within a certain time frame in a selected marketsegment. Since the query contains the *l_discount*, the *l_shipdate*, and the *o_shippriority*, Breadcrumb computes 12 schema alternatives for this scenario. To introduce errors to the query, we conduct two modifications to it that Breadcrumb has to find. First, we add a typo in the constant commitdate in one of the query's filters. Second, we replace the marketsegment in another filter. After these modifications, the query's result lacks a certain order. Therefore, the why-not question asks for this particular order.

Scenario Q10 computes customers who have returned items that have recently been ordered. For each customer, it further computes the revenue loss caused by the returns. The query references the attributes *l_discount*. Therefore, this scenario has two schema alternatives. We introduce three errors to the query that Breadcrumb has to find. We replace the constants in the filter on the returnflag and in the filter on the orderdate. Additionally, we substitute the discount with the tax in a projection that computes the discount on the correct, non-zero revenue. After these modifications, the query's result does not contain a customer who generates noticeable revenue. We define a why-not question that asks for this particular customer.

A detailed description of all scenarios with their schema alternatives, their queries formalized in our algebra, and their why-not question is available in [DLHG21b]. Their implementation is available on GitHub¹. Given the described workload, we discuss our experimental results, next.

8.4.2 Scalability with increasing dataset size

In the first experiment, we scale the Twitter and DBLP datasets from 100GB up to 500GB and measure Breadcrumbs runtime on the Twitter and DBLP scenarios with increasing dataset sizes. We run Breadcrumb RP and set a

¹<https://github.com/UniStuttgart-DataEngineering/breadcrumb>

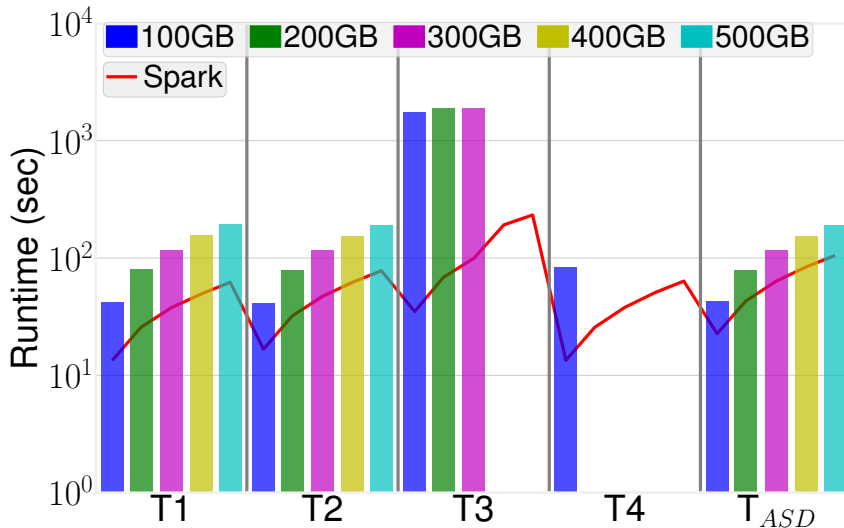


Figure 8.10: Runtime overhead on the Twitter dataset [DLHG21a]

time-out of two hours. This experiment serves to demonstrate Breadcrumb’s scalability on increasing dataset sizes.

Previous solutions that compute query-based explanations for missing answers have only been evaluated on floppy-disk-sized datasets [BHT15; CJ09; Her15]. Here, we report results on datasets that do not fit on 1000 floppy-disks to leave no doubt on Breadcrumb’s scalability.

We report the results for the Twitter scenarios in Figure 8.10 and for the DBLP scenarios in Figure 8.11. In both figures, the y-axis reports the runtime in seconds. It has a logarithmic scale. The x-axis shows the runtimes of each scenario with increasing dataset size. The line reports the queries’ runtimes when they are executed in plain Spark without Breadcrumb.

Across most scenarios (i.e., T1, T2, T_{ASD}, D1, D2, D3, D5) the runtimes scale linearly with the input size. In scenarios D4 and T4, the runtime does not grow linearly for the data sizes 100GB-300GB in T3 and 400GB-500GB in D4, respectively. In these scenarios, the increased annotation overhead causes Spark to spend most of the execution time with disk I/O, which masks Breadcrumb’s actual runtime overhead.

Depending on the scenario, Breadcrumb exceeds Spark’s runtime by a factor between 2.4 and 78.2. This overhead is in line with the reported overhead of state-of-the-art solutions on relational data. It is particularly

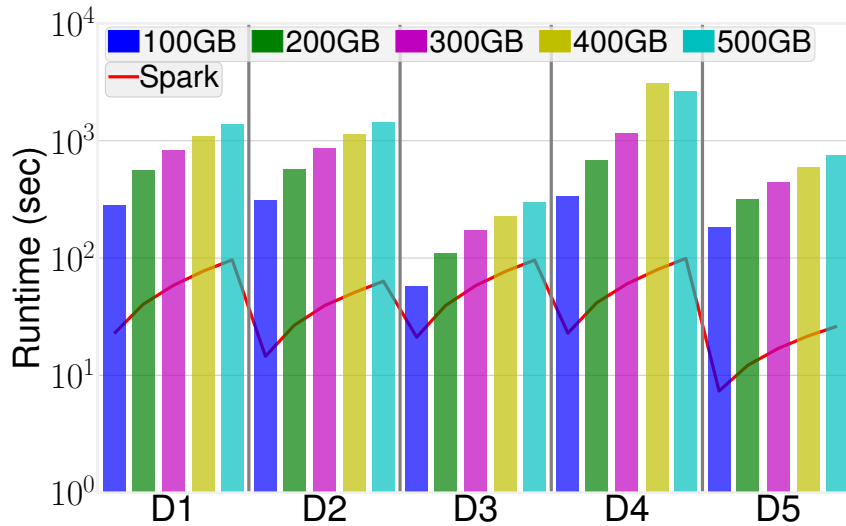


Figure 8.11: Runtime overhead on the DBLP dataset [DLHG21a]

low for scenarios whose queries have a low number of operators, such as D3, T2, and T_{ASD} . The overhead is generally higher and does not necessarily grow linearly any more when the queries become more complex (D4, D5, T3, T4). For such scenarios, Breadcrumb’s annotations grow in size with increasing dataset sizes. That causes additional runtime overhead. Hence, T3 exceeds the time-out limit for larger dataset sizes.

We can further derive detailed performance insight from this experiment. We observe the limitations on the join rewrite described in Section 7.2.4. The Hash-Joins in D4 and T3 become much slower Sort-Merge-Joins causing high overall runtime overhead regardless of the dataset size. Furthermore, we observe high runtime overhead when the scenario’s output is based on a small subset of the input tuples. For example, in D5, two inner flatten operators on two different nested relations that are empty for most tuples yield much fewer output tuples than input tuples. However, our tracing algorithm retains at least one output tuple for each input tuple. Finally, for T4, we only show results for 100GB input data because we did hit a Spark limitation for larger sizes. It is related to a reported bug in Spark’s grouping set implementation, which we use in the aggregation tracing procedure, and Spark’s current item limit in nested collections (2^{31}).

In summary, we have shown that Breadcrumb scales to large datasets in big data analytics systems. Even though Breadcrumb imposes an overhead of up to a factor 78.2 on the reported test scenarios, it outperforms all existing solutions by more than a factor 1000 regarding the input dataset size.

8.4.3 Scalability with multiple schema alternatives

We conduct two experiments to compare Breadcrumb's runtime with (RP) and without (RPnoSA) multiple schema alternatives. Since other state-of-the-art solutions do not support schema alternatives, these experiments are important to assess Breadcrumb's scalability regarding the number of schema alternatives. Each schema alternative imposes overhead on the runtime. These experiments quantify the overhead. The first experiment leverages the TPC-H scenarios since they have varying numbers of schema alternatives per scenario. This experiment exclusively compares the runtime overhead with schema alternatives enabled and disabled. To get a deeper understanding of the runtime impact of individual alternatives, we conduct a second experiment on select scenarios, in which we increment the number of alternatives one by one.

The results of the first experiment are shown in Figure 8.12. The bars in the figure describe the RP and RPnoSA runtimes for each scenario. For convenience, the figure also displays the number of schema alternatives when Breadcrumb runs in RP mode. In RPnoSA mode, the number always is 1 by definition. Note that the y-axis has a logarithmic scale. The line describes Spark's runtime without any Breadcrumb extensions. Within a single scenario, the time is the same for RP and RPnoSA.

The overhead between Spark and RPnoSA ranges between a factor of 3.9 and 10.1. Thus, it is significantly lower than the overhead imposed by RP, which grows up to a factor of 105.2. The overhead is generally higher than the overhead reported in the previous experiment because all TPC-H queries use aggregations. Thus, their result size is insignificant compared to the number of traced tuples (analogous to D5). When comparing RP with RPnoSA, we further note that increasing numbers of SAs cause higher

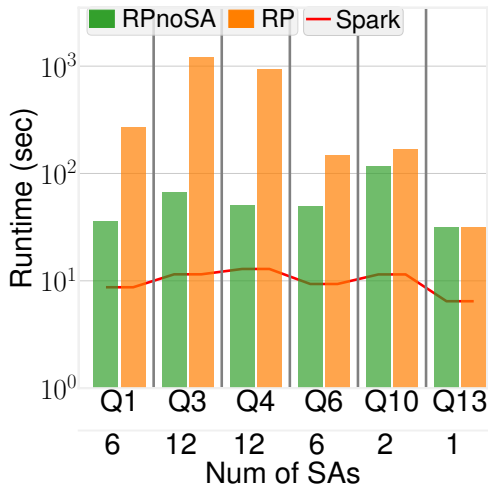


Figure 8.12: Runtime overhead on the TPC-H dataset [DLHG21a]

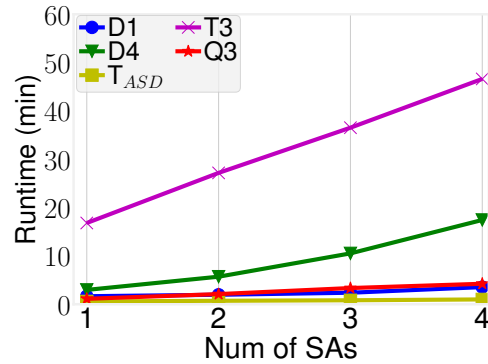


Figure 8.13: Runtime overhead with increasing number of schema alternatives [DLHG21a]

overheads. The relative overhead is highest in scenarios Q3 and Q4 since they have the highest number of schema alternatives. In Q13, the overhead does not grow between RPnoSA and RP because the scenario does not have additional schema alternatives. Hence, we conclude that running RP without additional schema alternatives does not impose additional overhead compared to RPnoSA. But the experiment has some caveats. While the experiment clearly indicates that the number of schema alternatives has runtime impact, it does not quantify the impact. Therefore, we conduct another series of measurements on select scenarios.

For the second experiment, we depict the scenarios D1, D4, T3, T_{ASD} since they run on different input datasets and have different query complexities. T_{ASD} represents the simple scenarios, D1 and T3 stand in for scenarios of intermediate complexity with relation flatten and join operators, and D4 and Q3 serve as representatives for complex scenarios that feature flatten, join, nesting, and aggregation operations. When we run the scenarios, we gradually increase the total number of schema alternatives from 1 to 4 and measure the runtime to understand how the number of schema alterna-






tives influences the runtime. Figure 8.13 shows the increasing number of alternatives on the x-axis and the runtime on the y-axis on a linear scale.

As Figure 8.13 illustrates, the runtime increases linearly in the scenarios of simple and intermediate complexity. More precisely, the runtime increases by the constant factor of 0.15 in T_{ASD} , 0.5 in D1, and 0.8 in T3 per added schema alternative. Since the factor is below 1, adding a schema alternative to the rewritten query is faster than executing separate queries for each alternative. In the complex scenarios, the runtime increases with a factor bigger than 1. Adding the first schema alternative to D4 yields a factor of 0.96. Adding the fourth alternative to the third alternative yields a factor of 1.47. Similarly, the factor grows beyond 1 in Q3, even though this is not clearly visible in the figure. Recall, however, that Q3 has up to 12 schema alternatives. When we run Q3 with 12 alternatives the runtime increases by a factor of 17.92 compared to the execution with just one alternative. We identify two root causes that explain factors beyond 1. First, with each added schema alternative, each tuple's size increases due to additional attributes referenced in the alternatives and additional provenance annotations. Second, Breadcrumb's grouping set implementation for relation nesting and aggregation operators (cf. Section 7.2.4) duplicates each input tuple for each alternative. Thus, both the tuple width and the tuple number increase with each SA. That explains the observed runtime factors that are larger than 1.

We derive two conclusions from the two experiments. First, the overhead imposed by schema alternatives is reasonable. Second, Breadcrumb scales linearly with an increasing number of schema alternatives, as long as the query pipelines do not contain aggregations or relation nestings. We conclude that the overhead imposed by schema alternatives is justifiable to get higher quality explanations. To reinforce this conclusion, we conduct another experiment that assesses the quality of the explanations.

8.4.4 Explanation quality

In this experiment, we evaluate Breadcrumb’s explanation quality. For that purpose, we provide an overview of the explanations computed by Breadcrumb with schema alternatives (RP), Breadcrumb without schema alternatives (RPnoSA), and WN++ for all scenarios. Recall that WN++ is the seminal Why-not algorithm [CJ09] extended to nested data and big data analytics systems. Afterward, we focus on the scenarios with a gold standard, namely T_{ASD} , and Q1 to Q13. We investigate whether the mentioned algorithms find the errors we have introduced into the scenarios’ queries. When the algorithms have found them, we also discuss the rank of the correct explanation in the returned list of explanations. The lower the rank, the better performs the algorithm. We have a detailed look at scenarios T_{ASD} , Q3, and Q10 before we derive general conclusions on the explanation quality.

Table 8.6 summarizes the explanations that WN++, Breadcrumb RPnoSA, Breadcrumb RP provide for all scenarios. Each row represents the scenario shown in the leftmost column. The six center columns in Table 8.6 describe the filter σ , projection π , join \bowtie , flatten F , nesting \mathcal{N} , and aggregation γ operator. If the cell is colored, the scenario’s query pipeline contains one or multiple instances of this operator. For instance, the pipeline in D1 contains all mentioned operators except from the aggregation. The pies in the cells indicate the operators that WN++, Breadcrumb RPnoSA, and Breadcrumb RP return as explanations. Pie  describes that all algorithms find an explanation involving the column’s operator type. The pie  denotes that WN++ misses an explanation involving an operator of the column’s type, which RPnoSA and RP find. If only RP identifies an operator as part of an explanation the related cell contains a pie . If WN++ produces an incomplete explanation that involves the found operator but requires an additional operator in the explanation, the cell is marked . When WN++ provides an operator as an explanation that is actually no explanation, the cell contains a pie . Note that some cells contain multiple pies because a query may contain multiple instances of the same operator type.

S	operators						# explanations		
	σ	π	\bowtie	F	\mathcal{N}	γ	WN++	RPnoSA	RP
D1	○	●					1	1	2
D2				●			0	0	1
D3					●		0	0	1
D4	○ ●			●			1	2	4
D5		●		○			1	1	2
T1	●			○ ● ●			1	1	2
T2	○ ●			●			1	2	4
T3				○ ●			1	1	2
T4	○ ●			●			1	1	3
T_{ASD}	●			●			0 (-)	0 (-)	2 (2)
Q1	○					●	1 (-)	1 (-)	3 (2)
Q3	○ ● ●					●	1 (-)	1 (1)	2 (1)
Q4	●					●	0 (-)	0 (-)	4 (3)
Q6	○ ● ●						1 (-)	7 (2)	11 (2)
Q10	○ ●	●	●				1(-)	2(-)	4 (4)
Q13			○				1 (1)	1(1)	1 (1)

○ : Found by all algorithms, ○● : found only by RPnoSA and RP, ●● : found only by RP, ○● : WN++ is incomplete ●● : WN++ is incorrect


Table 8.6: Summary of explanations returned for the lineage-based approach WN++, our reparameterization-based approach without SAs (RPnoSA) and our fully fledged approach RP. Shaded fields indicate that a scenario’s query uses one or more operators of this type, and the shaded circles indicate operators found by the different approaches (see legend) [DLHG21a].



The rightmost columns in Table 8.6 hold the number of returned explanations (not the number of operators in the explanations). The numbers in brackets behind the number of explanations indicate the position of the correct explanations according to their rank for the scenarios with a gold standard. Across all scenarios, WN++ finds 12, RPnoSA detects 21, and RP yields 48 explanations. WN++ finds at most one explanation in each scenario since it always returns only the last operator that removes a tuple that could potentially produce the missing answer. RP and RPnoSA often return multiple explanations that themselves contain non-distinct sets of

operators. In scenarios D2, D3, T_{ASD} , and Q4, RP is the only algorithm that provides an explanation. In these scenarios, the missing answer only appears in the result, when an attribute is exchanged with another attribute in the marked operator.

A closer look at the scenarios with a gold standard reveals that WN++ only finds one correct and complete explanation in scenario Q13. RPnoSA finds three out of seven correct explanations in scenarios Q3, Q6, and Q13. RP identifies the correct explanations in all seven scenarios. Further note that RP is the only algorithm that identifies non-selective operators, such as projections π , nestings \mathcal{N} , or aggregations γ as parts of explanations. To get a better understanding of the returned explanations, we analyze the scenarios T_{ASD} , Q3, and Q10 in more detail.

We start with the adaptive schema database scenario T_{ASD} . Recall that we miss a certain retweet in the query result. In our Twitter dataset, this tweet only appears in the retweet attribute. It does not occur in the quoted tweet attribute. Therefore, WN++ and RPnoSA do not provide any explanation. Only RP provides two explanations based on schema alternatives. RP returns the tuple flatten operator as the first explanation that flattens the quoted tweet attribute. The tweet appears in the result after reparameterizing the attribute in the flatten operator from the quoted tweet to the retweet attribute. Therefore, the explanation is syntactically correct. However, the explanation does not point to the two manipulated operators. As shown in Table 8.6, RP provides a second explanation, which contains the mentioned flatten operator and the filter operator that erroneously filters on the quoted count instead of the retweet count. This explanation precisely points to the two manipulated operators. Therefore, the second explanation also is semantically correct. In conclusion, this scenario shows that Breadcrumb finds explanations that lineage-based approaches such as WN++ miss since they do not support schema alternatives. Note that we can apply this observation to scenarios over nested data as well as flat relational data because exchanging attributes in operators is not limited to operators that exclusively manipulate or access nested data as the filter in this example shows.

Scenario Q3 computes unshipped orders and that we have manipulated two different filter operators. In the why-not question, we request a missing order. While RP provides two explanations, WN++ and RPnoSA return one explanation each. The latter two explanations are not the same, though. WN++ only returns the filter on the commitdate as an explanation since this filter removes the order entirely from the query's result. Unlike RPnoSA, WN++ misses that the succeeding filter on the marketsegment would also remove the missing order. Therefore, the corresponding cell on the filter operator in the Q3 row in Table 8.6 contains a  marker. RPnoSA and RP return both mentioned filters as the first and correct explanation. RP exclusively returns a second explanation that contains the final aggregation in addition to the two filters. It is derived from the schema alternatives that consider the tax as an alternative attribute to the discount. This explanation also yields the missing order since the order is part of the grouping attributes and not part of the aggregated attributes. In summary, this scenario shows that Breadcrumb outperforms WN++ even without schema alternatives.

Remember that scenario Q10 reports returned items and the associated revenue loss for each customer. We have introduced three errors in the scenario's query and request a customer with a non-zero revenue in the why-not question that is absent from the erroneous query result. WN++ returns the join operator on customer and order as an explanation because it removes the expected customer from the result. While the explanation makes the customer appear in the result, it cannot yield a non-zero revenue, which we explicitly ask for. Thus, this explanation is incorrect and marked with a pie  in Table 8.6. Breadcrumb does not return the join because it cannot yield a non-zero revenue. Unlike WN++, RPnoSA and RP provide the filter on the returnflag as the first explanation since the customer does not have any lineitems with the modified returnflag. Consequently, the filter removes all potential join partners for the expected customer. Therefore, a pie  appears in the filter column in Q10's row in Table 8.6. As the second explanation, RPnoSA and RP return both modified filters since the filter on the orderdate also removes tuples that join with the expected customer. Only RP returns two additional explanations. As the third explanation, RP

returns the filter on the returnflag together with the projection that computes the retail price of each returned lineitem. Finally, RP returns both filters with the mentioned projection. The last two explanations are based on schema alternatives. Thus, Table 8.6 holds two ● markers in Q10's row. The final explanation precisely points to the gold standard. It is ranked last since it modifies the most operators. However, one would have obtained the correct solution iteratively when observing the provided selections before the projection. We have observed similar possibilities in T_{ASD} . Hence, we summarize that observing the explanations with a higher rank helps to find and understand the correct explanation with a lower rank.

We conclude the evaluation of the explanation quality with the following general observations. Even RPnoSA provides explanations that WN++ misses (T1, T4, Q3, Q6, Q10) because RPnoSA traces tuples that potentially contribute to the missing result through the entire query. While the discussed results are obtained on nested data, WN++ exhibits the same problem for flat relational data. Furthermore, RP may find explanations based on schema alternatives that RP and WN++ miss (in all scenarios but Q13). As shown in D2, D3, T_{ASD} , and Q4, leveraging schema alternatives may be the only option to obtain any explanations. When multiple operators need reparameterizations, Breadcrumb provides the correct explanation, but possibly assigns it a low rank, like in Q10 and T_{ASD} . However, the operators in higher ranked explanations typically overlap with the operators in the correct explanation. Thus, starting investigations with the higher-ranked explanations seems a viable option to incrementally correct a query pipeline.

8.4.5 Discussion

In this section, we have shown that Breadcrumb runs on nested datasets that are more than a factor 1000 larger than the flat relational datasets used to evaluate other state-of-the-art solutions. Even on these large datasets, Breadcrumb scales with increasing dataset sizes as long as sufficient compute resources are available. Furthermore, we have observed that Breadcrumb scales with an increasing number of schema alternatives that other solutions

do not even support. Thanks to these alternatives, Breadcrumb finds more explanations than other existing approaches. Even without schema alternatives, Breadcrumb yields more accurate explanations than the why-not algorithm [CJ09] because it traces the tuples that potentially contribute to the missing result through the entire query.

8.5 Summary

In this chapter, we have addressed our Contributions (2) to (5). We have shown that the tree-pattern matching algorithm, Pebble, and Breadcrumb scale to large datasets in big data analytics systems. Further, algorithm-specific experiments underline the algorithms' scalability in varying aspects. Notably, Pebble and Breadcrumb are the first algorithms that compute provenance-based explanations on 100+ GB nested data and the tree-pattern matching algorithm even outperforms plain Spark in some cases.

Additionally, the tree-pattern matching algorithm simplifies the query pipeline definition compared to plain Spark. Pebble and Breadcrumb provide more comprehensive explanations than comparable state-of-the-art algorithms. We have illustrated the need for these comprehensive explanations in real-world use-cases beyond debugging.

CONCLUSION & OUTLOOK

Big data analytics systems, such as Apache Spark, Flink, or Hive support nested data and scale with increasing computing resources. Thus, they are frequently means of choice to process large amounts of heterogeneous data. However, the systems have very limited means to systematically debug analytical query pipelines that process the nested data.

In this context, we have studied the research question of how to explain existing and missing query results in big data analytics systems when processing nested data. We have addressed this question by contributing novel means to provide explanations for existing and missing data in the result of big data analytics queries that process nested data. We have studied the problem from the ground up. First, we have defined a nested data model and nested relational algebra which closely resembles the provided data formats and operators in big data analytics systems. Second, we have contributed distributed tree-pattern matching to define and retrieve data to be explained. In a more general context, the tree-pattern matching enables the addressing and accessing of individual nested data in big data analytics queries. Third, we have provided novel explanations for existing data in the result of big data analytics queries. Fourth, we have contributed explanations for missing data

in the result of the analytics queries. Both explanation types help to debug complex analytical queries over nested data. Finally, we have implemented the concepts on top of the big data analytics system and evaluated them on two nested real-world datasets of up to 500GB to show that they scale to practically meaningful datasets. We summarize them here and provide an outlook on future work.

9.1 Conclusion

We have defined a nested data model and a nested relational algebra that faithfully capture the nested data formats and the execution semantics of big data analytics systems [DLHG21a; DLHG21b]. They are necessary to get explanations that correspond directly to the queries defined over nested data in big data analytics systems. At the same time, the models allow us to abstract from a particular big data analytics system and to express generally applicable concepts and algorithms. All the following contributions rely on the nested data model and a nested relational algebra.

To request explanations for selected existing or missing data and to concisely address nested data during query processing, we have proposed tree-patterns and contributed a tree-pattern matching algorithm that is tailored to big data analytics systems [DH20a]. The algorithm leverages the following key feature of these systems to scale to large datasets. The systems process data that share a common schema. Hence, it can split the matching into two phases to avoid global state. In its first phase, it computes schema matches and applies these matches on the data in the second phase. Leveraging the matches, the algorithm directly accesses nested data that is referenced in the tree-pattern. It does not scrape each individual data value as many other existing algorithms do. With experiments on two real-world workload we have shown that the algorithm scales with increasing dataset sizes and increasing compute resources. Furthermore, we have experimentally validated that the schema-matching phase of the tree-pattern matching algorithm has data-instance independent runtime. Moreover, we have shown that query

pipelines with tree-patterns are less complex and potentially faster than the same pipelines without tree-patterns.

We have further introduced the novel structural provenance to provide explanations for existing data [DH19; DH20b]. These explanations are more comprehensive than those of other solutions since they capture the access to and the manipulation of the data at the granularity of individual nested attributes. To precisely collect the structural provenance, we have extended each operator in our nested relational algebra to capture manipulations and access to the data's values and structure. Since the structural provenance stores a lot of redundant data when captured precisely according to the extended rules, we have introduced the lightweight structural provenance. It captures structural manipulations on the schema and data dependencies on the instance to minimize the cost of collecting the same. It allows the Pebble algorithm to efficiently capture the structural provenance during query execution. Based on the lightweight provenance, Pebble computes fully-fledged structural provenance explanations for existing data during backtracing. We have conducted experiments on datasets that are 100+ times larger than the ones used to evaluate other existing provenance solutions that support nested data. These experiments show that Pebble scales with increasing dataset sizes because Pebble barely imposes more space and runtime overhead on the query execution than solutions that trace provenance at the coarser-grained granularity of top-level tuples. The experiments finally show that Pebble's rich explanations enable novel use-cases beyond debugging, such as finding data-usage patterns and auditing.

Moreover, we have contributed a reparameterization-based approach to provide query-based explanations for missing answers in the result [DGHL19; DLGH21; DLHG21a; DLHG21b]. To the best of our knowledge, it is the first approach that supports nested data, which conforms to our data model. Furthermore, we have formally introduced the novel concept of reparameterizations based on our algebra to account for errors in the query that existing solutions miss, such as misinterpreted attributes. Existing solutions compute query-based explanations exclusively from provenance. They are limited to finding operators that remove data from their input, such as se-

lections and joins. Since our solution extends the existing solutions with reparameterizations, it can find explanations that contain any operator type with parameters, including projections, aggregations, or relation nesting operators. Based on these reparameterizations, we have formally introduced query-based explanations. These explanations consider the reparameterized operators and the side-effects on the result to prevent unnecessary manipulations of the query and excessive changes to the result. Since computing these explanations is NP-hard in the general case and remains computationally infeasible if we restrict the set of operators, we have proposed the heuristic Breadcrumb algorithm. It applies two novel techniques to efficiently approximate the explanations given nested input data. It re-validates each intermediate result to precisely identify the data that potentially contribute to an explanation. That is particularly important when nested data are involved in the query. Furthermore, it leverages schema alternatives to group similar reparameterizations together and execute them simultaneously. That is a prerequisite to scale to large datasets. Our experiments have shown that Breadcrumb provides more explanations than existing state-of-the-art solutions even on flat data. These explanations are more comprehensive than those of existing solutions since they consider reparameterizations rather than mere provenance. The experiments have further shown that Breadcrumb scales to datasets that are 1000+ times larger than those used to evaluate other solutions that compute query-based explanation.

9.2 Outlook

Based on the research contributions presented in this work, we propose several avenues of future work. The concepts and algorithms introduced in this work target big data analytics systems that process large, nested datasets. With this knowledge in mind, the following topics are worth looking into in future research.

Extensions to further operators occurring in analytical queries. While the systems already support a rich set of operators, analytical queries in big

data analytics systems typically rely on a richer set of operators. In particular, it would be interesting to extend Pebble and Breadcrumb with support for top-k queries and window functions since they frequently occur in complex analytical queries.

Tighter bounds for the side-effect estimation in Breadcrumb. Breadcrumb's current implementation leverages very loose side-effect estimations to identify minimal successful reparameterizations. A formal investigation of tighter bounds for the side-effects allows for more effective pruning of the returned explanations. That accelerates the debugging process.

Novel applications and use-cases. In the context of Pebble, we have started investigating novel applications beyond debugging for Pebble's explanations. We are convinced that Pebble and Breadcrumb enable further use-cases and applications that are worth looking into in the future. They may be leveraged for (partial) view updates, query optimizations, or data layout optimizations.

More generally, the contributions presented in this work are a cornerstone for a rich set of novel research topics related to tracing nested data in big data analytics queries. In particular, these systems have initially been designed with debugging complex queries in mind. During this work, we have encountered a wide variety of further applications in the direction of data layout and query optimization that would be interesting to investigate thoroughly in the future.

LIST OF PUBLICATIONS

- [1] R. Diestelkämper, S. Lee, B. Glavic, M. Herschel. “Debugging Missing Answers for Spark Queries over Nested Data with Breadcrumb.” In: *Proceedings of the VLDB Endowment (PVLDB)* (2021), pp. 1–4.
- [2] R. Diestelkämper, S. Lee, M. Herschel, B. Glavic. “To not miss the forest for the trees – a holistic approach for explaining missing answers over nested data.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2021, pp. 1–13.
- [3] R. Diestelkämper, S. Lee, M. Herschel, B. Glavic. *To not miss the forest for the trees – a holistic approach for explaining missing answers over nested data (extended version)*. 2021. arXiv: 2103.07561 [cs.DB].
- [4] R. Diestelkämper, M. Herschel. “Distributed Tree-Pattern Matching in Big Data Analytics Systems.” In: *Advances in Databases and Information Systems (ADBIS)*. 2020, pp. 171–186.
- [5] R. Diestelkämper, M. Herschel. “Tracing nested data with structural provenance for big data analytics.” In: *Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2020, pp. 253–264.
- [6] R. Diestelkämper, B. Glavic, M. Herschel, S. Lee. “Query-Based Why-Not Explanations for Nested Data.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. USENIX Association, 2019, pp. 1–4.
- [7] R. Diestelkämper, M. Herschel. “Capturing and Querying Structural Provenance in Spark with Pebble.” In: *ACM Conference on the Management of Data (SIGMOD)*. ACM, 2019, pp. 1893–1896.

- [8] R. Diestelkämper, M. Herschel, P. Jadhav. “Provenance in DISC Systems: Reducing Space Overhead at Runtime.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. 2017, pp. 1–5.
- [9] M. Herschel, R. Diestelkämper, H. Ben Lahmar. “A survey on provenance: What for? What form? What from?” In: *The VLDB Journal* 26.6 (2017), pp. 881–906.

BIBLIOGRAPHY

- [ABC+10] U. Acar, P. Buneman, J. Cheney, J. Van Den Bussche, N. Kwasnikowska, S. Vansummeren. “A Graph Model of Data and Workflow Provenance.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. 2010, pp. 1–8 (cit. on p. 38).
- [ACLS01] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, D. Srivastava. “Minimization of Tree Pattern Queries.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2001, pp. 497–508 (cit. on p. 33).
- [ADD+11] Y. Amsterdamer, S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, V. Tannen. “Putting Lipstick on Pig: Enabling Database-style Workflow Provenance.” In: *Proceedings of the VLDB Endowment (PVLDB)* 5.4 (2011), pp. 346–357 (cit. on pp. 16, 36, 38, 40, 42, 208, 218, 219).
- [ADMR05] D. Aumueller, H.-H. Do, S. Massmann, E. Rahm. “Schema and Ontology Matching with COMA++.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2005, pp. 906–908 (cit. on p. 147).
- [ADT11] Y. Amsterdamer, D. Deutch, V. Tannen. “Provenance for aggregate queries.” In: *ACM Symposium on Principles of Database Systems (PODS)*. 2011, pp. 153–164 (cit. on p. 38).
- [AFG+18] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, Q. Zeng. “GProM - A Swiss Army Knife for Your Provenance Needs.” In: *IEEE Data Engineering Bulletin* 41.1 (2018), pp. 51–62 (cit. on p. 16).
- [AJK+02] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. Patel, D. Srivastava, Y. Wu. “Structural joins: a primitive for efficient XML query pattern matching.”

- In: *IEEE International Conference on Data Engineering (ICDE)*. 2002, pp. 141–152 (cit. on p. 34).
- [ASH13] S. Akoush, R. Sohan, A. Hopper. “HadoopProv: Towards Provenance As a First Class Citizen in MapReduce.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. 2013, pp. 1–4 (cit. on p. 36).
- [AXL+15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, M. Zaharia. “Spark SQL: Relational Data Processing in Spark.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2015, pp. 1383–1394 (cit. on p. 179).
- [Bar19] P. Barceló. “A Theoretical View on Reverse Engineering Problems for Database Query Languages.” In: *International Workshop on Description Logics*. 2019, pp. 1–2 (cit. on p. 46).
- [Bel18] K. Belhajjame. “On Answering Why-Not Queries Against Scientific Workflow Provenance.” In: *Conference on Extending Database Technology (EDBT)*. 2018, pp. 465–468 (cit. on p. 45).
- [BHT14] N. Bidoit, M. Herschel, K. Tzompanaki. “Query-Based Why-Not Provenance with NedExplain.” In: *Conference on Extending Database Technology (EDBT)*. 2014, pp. 145–156 (cit. on pp. 17, 45, 136).
- [BHT15] N. Bidoit, M. Herschel, A. Tzompanaki. “Efficient Computation of Polynomial Explanations of Why-Not Questions.” In: *Conference on Information and Knowledge Management (CIKM)*. 2015, pp. 713–722 (cit. on pp. 17, 45, 136, 224).
- [Bil05] P. Bille. “A survey on tree edit distance and related problems.” In: *Theoretical computer science 337.1-3 (2005)*, pp. 217–239 (cit. on p. 139).
- [BKS02] N. Bruno, N. Koudas, D. Srivastava. “Holistic Twig Joins: Optimal XML Pattern Matching.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2002, pp. 310–321 (cit. on pp. 34, 91).
- [BKT01] P. Buneman, S. Khanna, W. C. Tan. “Why and Where: A Characterization of Data Provenance.” In: *International Conference on Database Theory (ICDT)*. 2001, pp. 316–330 (cit. on p. 38).

- [BNTW95] P. Buneman, S. Naqvi, V. Tannen, L. Wong. “Principles of programming with complex objects and collection types.” In: *Theoretical Computer Science* 149.1 (1995), pp. 3–48 (cit. on p. 39).
- [CAA14] J. Cheney, A. Ahmed, U. A. Acar. “Database Queries That Explain Their Work.” In: *Symposium on Principles and Practice of Declarative Programming (PDPP)*. 2014, pp. 271–282 (cit. on pp. 16, 39, 40).
- [CCT09] J. Cheney, L. Chiticariu, W.-C. Tan. “Provenance in Databases: Why, How, and Where.” In: *Foundations and Trends in Databases* 1.4 (2009), pp. 379–474 (cit. on p. 38).
- [CJ09] A. Chapman, H. V. Jagadish. “Why not?” In: *ACM Conference on the Management of Data (SIGMOD)*. 2009, pp. 523–534 (cit. on pp. 17, 44, 45, 136, 220, 224, 229, 234).
- [CLMR16] Z. Chothia, J. Liagouris, F. McSherry, T. Roscoe. “Explaining outputs in modern data analytics.” In: *Proceedings of the VLDB Endowment (PVLDB)* 9.12 (2016), pp. 1137–1148 (cit. on p. 36).
- [DAB16] G. Diaz, M. Arenas, M. Benedikt. “Sparqlbye: Querying RDF data by example.” In: *Proceedings of the VLDB Endowment (PVLDB)* 9.13 (2016), pp. 1533–1536 (cit. on p. 46).
- [DFGH18] D. Deutch, N. Frost, A. Gilad, T. Haimovich. “NLProveNAns: Natural Language Provenance for Non-Answers.” In: *Proceedings of the VLDB Endowment (PVLDB)* 11.12 (2018), pp. 1986–1989 (cit. on pp. 45, 136).
- [DFGH20] D. Deutch, N. Frost, A. Gilad, T. Haimovich. “Explaining Missing Query Results in Natural Language.” In: *Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2020, pp. 427–430 (cit. on p. 45).
- [DG19] D. Deutch, A. Gilad. “Reverse-Engineering Conjunctive Queries from Provenance Examples.” In: *Conference on Extending Database Technology (EDBT)*. 2019, pp. 277–288 (cit. on p. 46).
- [DGHL19] R. Diestelkämper, B. Glavic, M. Herschel, S. Lee. “Query-Based Why-Not Explanations for Nested Data.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. USENIX Association, 2019, pp. 1–4 (cit. on pp. 21, 31, 132, 207, 208, 237).

- [DH19] R. Diestelkämper, M. Herschel. “Capturing and Querying Structural Provenance in Spark with Pebble.” In: *ACM Conference on the Management of Data (SIGMOD)*. ACM, 2019, pp. 1893–1896 (cit. on pp. 19, 36, 94, 237).
- [DH20a] R. Diestelkämper, M. Herschel. “Distributed Tree-Pattern Matching in Big Data Analytics Systems.” In: *Advances in Databases and Information Systems (ADBIS)*. 2020, pp. 171–186 (cit. on pp. 19, 31, 74, 81, 179, 199, 200, 202, 203, 205, 236).
- [DH20b] R. Diestelkämper, M. Herschel. “Tracing nested data with structural provenance for big data analytics.” In: *Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2020, pp. 253–264 (cit. on pp. 18, 19, 31, 36, 41, 42, 51, 94, 179, 206, 207, 209, 210, 212, 215–217, 237).
- [DLGH21] R. Diestelkämper, S. Lee, B. Glavic, M. Herschel. “Debugging Missing Answers for Spark Queries over Nested Data with Breadcrumb.” In: *Proceedings of the VLDB Endowment (PVLDB) (2021)*, pp. 1–4 (cit. on pp. 21, 31, 132, 237).
- [DLHG21a] R. Diestelkämper, S. Lee, M. Herschel, B. Glavic. “To not miss the forest for the trees – a holistic approach for explaining missing answers over nested data.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2021, pp. 1–13 (cit. on pp. 18, 21, 31, 51, 59, 132, 149, 179, 220, 224, 225, 227, 230, 236, 237).
- [DLHG21b] R. Diestelkämper, S. Lee, M. Herschel, B. Glavic. *To not miss the forest for the trees – a holistic approach for explaining missing answers over nested data (extended version)*. 2021. arXiv: 2103.07561 [cs.DB] (cit. on pp. 18, 21, 31, 51, 132, 143, 179, 220, 221, 223, 236, 237).
- [DR02] H. Do, E. Rahm. “COMA - A System for Flexible Combination of Schema Matching Approaches.” In: *Conference on Very Large Data Bases (VLDB)*. 2002, pp. 610–621 (cit. on p. 147).
- [FGT08] J. N. Foster, T. Green, V. Tannen. “Annotated XML: queries and provenance.” In: *ACM Symposium on Principles of Database Systems (PODS)*. 2008, pp. 271–280 (cit. on pp. 16, 38).

- [FKSS08] J. Freire, D. Koop, E. Santos, C. T. Silva. “Provenance for Computational Tasks: A Survey.” In: *Computing in Science and Engineering* 10.3 (2008), pp. 11–21 (cit. on p. 36).
- [GBH10] N. Grimsmo, T. Bjørklund, M. Hetland. “Fast Optimal Twig Joins.” In: *Proceedings of the VLDB Endowment (PVLDB)* 3.12 (2010), pp. 894–905 (cit. on p. 34).
- [GIY+16] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, M. Kim. “BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark.” In: *International Conference on Software Engineering (ICSE)*. 2016, pp. 784–795 (cit. on p. 16).
- [Gla21] B. Glavic. “Data Provenance - Origins, Applications, Algorithms, and Models.” In: *Foundations and Trends in Databases* 9.3-4 (2021), pp. 209–441 (cit. on pp. 36, 39).
- [GM93] S. Grumbach, T. Milo. “Towards Tractable Algebras for Bags.” In: *ACM Symposium on Principles of Database Systems (PODS)*. 1993, pp. 49–58 (cit. on pp. 51, 59).
- [GMF+20] T. Guedes, L. B. Martins, M. L. F. Falci, V. Silva, K. A. Ocana, M. Matoso, M. Bedo, D. de Oliveira. “Capturing and Analyzing Provenance from Spark-based Scientific Workflows with SAMBA-RaP.” In: *Future Generation Computer Systems* 112 (2020), pp. 658–669 (cit. on pp. 36, 37, 42).
- [GSM+18] T. Guedes, V. Silva, M. Mattoso, M. Bedo, D. de Oliveira. “A Practical Roadmap for Provenance Capture and Data Analysis in Spark-Based Scientific Workflows.” In: *Workflows in Support of Large-Scale Science (WORKS)*. 2018, pp. 31–41 (cit. on pp. 36, 37, 42).
- [HCDN08] J. Huang, T. Chen, A. Doan, J. F. Naughton. “On the provenance of non-answers to queries over extracted data.” In: *Proceedings of the VLDB Endowment (PVLDB)* 1.1 (2008), pp. 736–747 (cit. on p. 44).
- [HD13] M. Hachicha, J. Darmont. “A Survey of XML Tree Patterns.” In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 25.1 (2013), pp. 29–46 (cit. on pp. 16, 31–33).

- [HDB17] M. Herschel, R. Diestelkämper, H. Ben Lahmar. “A survey on provenance: What for? What form? What from?” In: *The VLDB Journal* 26.6 (2017), pp. 881–906 (cit. on pp. 16, 31, 35, 36, 44, 136).
- [Her15] M. Herschel. “A Hybrid Approach to Answering Why-Not Questions on Relational Query Results.” In: *ACM Journal on Data and Information Quality (JDIQ)* 5.3 (2015), pp. 1–29 (cit. on pp. 44, 45, 136, 224).
- [HH10] M. Herschel, M. A. Hernández. “Explaining Missing Answers to SPJUA Queries.” In: *Proceedings of the VLDB Endowment (PVLDB)* 3.1 (2010), pp. 185–196 (cit. on pp. 17, 44).
- [HHT09] M. Herschel, M. A. Hernández, W. C. Tan. “Artemis: A System for Analyzing Missing Answers.” In: *Proceedings of the VLDB Endowment (PVLDB)* 2.2 (2009), pp. 1550–1553 (cit. on pp. 17, 44).
- [IES+18] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein, T. Condie. “Adding data provenance support to Apache Spark.” In: *The VLDB Journal* 27.5 (2018), pp. 595–615 (cit. on p. 36).
- [IHH09] S. K. Izadi, T. Härder, M. Haghjoo. “S3: Evaluation of tree-pattern XML queries supported by structural summaries.” In: *Data & Knowledge Engineering* 68.1 (2009), pp. 126–145 (cit. on p. 34).
- [ILZ14] M. S. Islam, C. Liu, R. Zhou. “Flexiq: A flexible interactive querying framework by exploiting the skyline operator.” In: *Journal of Systems and Software* 97 (2014), pp. 97–117 (cit. on p. 17).
- [IPW11] R. Ikeda, H. Park, J. Widom. “Provenance for Generalized Map and Reduce Workflows.” In: *Conference on Innovative Data Systems Research (CIDR)*. 2011, pp. 273–283 (cit. on pp. 16, 36, 42, 43, 117, 211).
- [IST+15] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, T. Condie. “Titian: Data Provenance Support in Spark.” In: *Proceedings of the VLDB Endowment (PVLDB)* 9.3 (2015), pp. 216–227 (cit. on pp. 16, 36, 42, 43, 117, 190, 206, 211, 215, 218).
- [KG12] G. Karvounarakis, T. J. Green. “Semiring-Annotated Data: Queries and Provenance.” In: *SIGMOD Record* 41.3 (2012), pp. 5–14 (cit. on p. 38).
- [KLS18] D. V. Kalashnikov, L. V. Lakshmanan, D. Srivastava. “FastQRE: Fast Query Reverse Engineering.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2018, pp. 337–350 (cit. on p. 46).

- [KLZ13] S. Köhler, B. Ludäscher, D. Zinn. “First-Order Provenance Games.” In: *In Search of Elegance in the Theory and Practice of Computation*. 2013, pp. 382–399 (cit. on p. 44).
- [KV08] N. Kwasnikowska, J. Van den Bussche. “Mapping the NRC Dataflow Model to the Open Provenance Model.” In: *Workshop on Theory and Practice of Provenance (TAPP)*. 2008, pp. 3–16 (cit. on p. 38).
- [LCL04] J. Lu, T. Chen, T. W. Ling. “Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach.” In: *Conference on Information and Knowledge Management (CIKM)*. 2004, pp. 533–542 (cit. on p. 34).
- [LDY13] D. Logothetis, S. De, K. Yocum. “Scalable lineage capture for debugging DISC analytics.” In: *Symposium on Cloud Computing (SoCC)*. 2013, pp. 1–15 (cit. on pp. 16, 36, 42, 43, 117, 211).
- [Ley09] M. Ley. “DBLP - Some Lessons Learned.” In: *Proceedings of the VLDB Endowment (PVLDB) 2.2* (2009), pp. 1493–1500 (cit. on p. 198).
- [LKL17] S. Lee, S. Köhler, B. Ludäscher, B. Glavic. “A SQL-Middleware Unifying Why and Why-Not Provenance for First-Order Queries.” In: *IEEE International Conference on Data Engineering (ICDE)*. 2017, pp. 485–496 (cit. on pp. 17, 44).
- [LLBW11] J. Lu, T. Ling, Z. Bao, C. Wang. “Extended XML Tree Pattern Matching: Theories and Algorithms.” In: *IEEE Transactions on Knowledge and Data Engineering (TKDE) 23.3* (2011), pp. 402–416 (cit. on p. 31).
- [LLCC05] J. Lu, T. Ling, C.-Y. Chan, T. Chen. “From region encoding to extended Dewey: On efficient processing of XML twig pattern matching.” In: *Conference on Very Large Data Bases (VLDB)*. 2005, pp. 193–204 (cit. on pp. 34, 82, 91, 181).
- [LLG20] S. Lee, B. Ludäscher, B. Glavic. “Approximate Summaries for Why and Why-not Provenance.” In: *Proceedings of the VLDB Endowment (PVLDB) 13.6* (2020), pp. 912–924 (cit. on pp. 17, 44).
- [LML11] J. Lu, X. Meng, T. W. Ling. “Indexing and Querying XML Using Extended Dewey Labeling Scheme.” In: *Data & Knowledge Engineering 70.1* (2011), pp. 35–59 (cit. on p. 34).

- [MBC13] P. Missier, K. Belhajjame, J. Cheney. “The W3C PROV family of specifications for modelling provenance metadata.” In: *Conference on Extending Database Technology (EDBT)*. 2013, pp. 773–776 (cit. on p. 16).
- [MDG18] T. Müller, B. Dietrich, T. Grust. “You Say ‘What’, I Hear ‘Where’ and ‘Why’? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance.” In: *Proceedings of the VLDB Endowment (PVLDB)* 11.11 (2018), pp. 1536–1549 (cit. on p. 16).
- [MGMS10] A. Meliou, W. Gatterbauer, K. F. Moore, D. Suciu. “The Complexity of Causality and Responsibility for Query Answers and non-Answers.” In: *Proceedings of the VLDB Endowment (PVLDB)* 4.1 (2010), pp. 34–45 (cit. on pp. 17, 44).
- [MK09] C. Mishra, N. Koudas. “Interactive Query Refinement.” In: *Conference on Extending Database Technology (EDBT)*. 2009, pp. 862–873 (cit. on p. 47).
- [MKZ08] C. Mishra, N. Koudas, C. Zuzarte. “Generating Targeted Queries for Database Testing.” In: *ACM Conference on the Management of Data (SIGMOD)*. New York, NY, USA, 2008, pp. 499–510 (cit. on p. 47).
- [MMR+16] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, Y. Velegarakis. “A Holistic and Principled Approach for the Empty-answer Problem.” In: *The VLDB Journal* 25.4 (2016), pp. 597–622 (cit. on p. 47).
- [Mor10] L. Moreau. “The Foundations for Provenance on the Web.” In: *Foundations and Trends in Web Science* 2.2-3 (2010), pp. 99–241 (cit. on p. 36).
- [MS02] G. Miklau, D. Suciu. “Containment and Equivalence for an XPath Fragment.” In: *ACM Symposium on Principles of Database Systems (PODS)*. 2002, pp. 65–76 (cit. on p. 33).
- [OR11] C. Olston, B. Reed. “Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows.” In: *Proceedings of the VLDB Endowment (PVLDB)* 4.12 (2011), pp. 1237–1248 (cit. on p. 36).
- [PA11] M. Pawlik, N. Augsten. “RTED: a robust algorithm for the tree edit distance.” In: *Proceedings of the VLDB Endowment (PVLDB)* 5.4 (2011), pp. 334–345 (cit. on p. 139).

- [PCW17] P. Pirzadeh, M. Carey, T. Westmann. “A performance study of big data analytics platforms.” In: *Conference on Big Data*. 2017, pp. 2911–2920 (cit. on pp. 197, 221).
- [PFMB19] J. F. Pimentel, J. Freire, L. Murta, V. Braganholo. “A Survey on Collecting, Managing, and Analyzing Provenance from Scripts.” In: *ACM Computing Surveys* 52.3 (2019), pp. 1–38 (cit. on p. 36).
- [RESC15] E. D. Ragan, A. Endert, J. Sanyal, J. Chen. “Characterizing Provenance in Visualization and Data Analysis: An Organizational Framework of Provenance Types and Purposes.” In: *IEEE Transactions on Visualization and Computer Graphics*. 2015, pp. 31–40 (cit. on p. 36).
- [SAC+17] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, Y. Yang. “Adaptive Schema Databases.” In: *Conference on Innovative Data Systems Research (CIDR)*. 2017 (cit. on pp. 221, 222).
- [TC10] Q. T. Tran, C.-Y. Chan. “How to ConQueR why-not questions.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2010, pp. 15–26 (cit. on pp. 17, 45).
- [TCP14] Q. T. Tran, C. Y. Chan, S. Parthasarathy. “Query Reverse Engineering.” In: *The VLDB Journal* 23.5 (2014), pp. 721–746 (cit. on p. 46).
- [TPL+13] M. Tahraoui, K. Pinel-Sauvagnat, C. Laitang, M. Boughanem, H. Kheddouci, L. Ning. “A survey on tree matching and XML retrieval.” In: *Computer Science Review* 8 (2013), pp. 1–23 (cit. on p. 31).
- [Tra09] Transaction Processing Council. *TPC-H Benchmark Specification*. 2009 (cit. on pp. 197, 221, 222).
- [TTT19] M. T. Tchendji, L. Taddonfouet, T. T. Tchendji. “A Tree Pattern Matching Algorithm for XML Queries with Structural Preferences.” In: *Journal of Computer and Communications* 7 (2019), pp. 61–83 (cit. on p. 31).
- [TZES17] W. C. Tan, M. Zhang, H. Elmeleegy, D. Srivastava. “Reverse Engineering Aggregation Queries.” In: *Proceedings of the VLDB Endowment (PVLDB)* 10.11 (2017), pp. 1394–1405 (cit. on p. 46).
- [WC17] Z. Wang, S. Chen. “Exploiting Common Patterns for Tree-Structured Data.” In: *ACM Conference on the Management of Data (SIGMOD)*. ACM, 2017, pp. 883–896 (cit. on p. 198).

- [WL08] X. Wu, G. Liu. “XML twig pattern matching using version tree.” In: *Data & Knowledge Engineering* 64.3 (2008), pp. 580–599 (cit. on p. 34).
- [ZAI19] N. Zheng, A. Alawini, Z. G. Ives. “Fine-Grained Provenance for Matching and ETL.” In: *IEEE International Conference on Data Engineering (ICDE)*. 2019, pp. 184–195 (cit. on pp. 16, 38, 40, 42, 208, 214, 218, 219).
- [Zlo77] M. Zloof. “Query-by-Example: A Data Base Language.” In: *IBM Systems Journal* 16.4 (1977), pp. 324–343 (cit. on p. 46).
- [ZND+01] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. “On Supporting Containment Queries in Relational Database Management Systems.” In: *ACM Conference on the Management of Data (SIGMOD)*. 2001, pp. 425–436 (cit. on p. 34).
- [ZSS92] K. Zhang, R. Statman, D. Shasha. “On the editing distance between unordered labeled trees.” In: *Information processing letters* 42.3 (1992), pp. 133–139 (cit. on pp. 139, 171).
- [ZXW+16] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker. “Apache Spark: A Unified Engine for Big Data Processing.” In: *Communications of the ACM (CACM)* 59.11 (2016), pp. 56–65 (cit. on p. 179).

URLs were last checked on May 23rd, 2021.

LIST OF FIGURES

1.1	Example query pipeline	22
1.2	Tree-patterns	25
3.1	First tuple in Table 1.1 represented as a tree	57
3.2	Schema of the tuples in Table 1.1 represented as a tree	57
3.3	System architecture	68
4.1	Example tree-pattern T	76
4.2	Schema matching example	80
4.3	Example DeweyIDs	83
5.1	Example tree-pattern T	115
5.2	Example provenance tree \mathcal{PT}_{ex}	115
5.3	Example provenance tree $\mathcal{PT}_{preNest}$	123
5.4	Example provenance tree $\mathcal{PT}_{postFlatten}$	125
5.5	Provenance trees \mathcal{PT} on the input relation	125
6.1	Example execution pipeline	140
6.2	Enumerating and pruning schema alternatives	149
8.1	Runtime of the schema-matching	201

8.2	Runtime of the tree-pattern matching algorithm	202
8.3	Runtime with tree-pattern matching (TPM) and without (Spark)	203
8.4	Runtime overhead on Twitter dataset	209
8.5	Runtime overhead on DBLP dataset	210
8.6	Size of collected provenance	212
8.7	Runtime of Pebble’s backtracing	215
8.8	Runtime comparison Titian	216
8.9	HeatMap for 25 tuples in the DBLP inproceedings dataset . .	217
8.10	Runtime overhead on the Twitter dataset	224
8.11	Runtime overhead on the DBLP dataset	225
8.12	Runtime overhead on the TPC-H dataset	227
8.13	Runtime overhead with increasing number of schema alter- natives	227

LIST OF TABLES

1.1	Example input data	23
1.2	Example output data	23
1.3	Unexpected data that is existing in the example output	24
1.4	Expected data that is missing in the example output	24
1.5	Explanations for the existing example output	26
1.6	Explanations for the missing example output	27
2.1	Overview of big data analytics provenance solutions	36
2.2	Feature overview of provenance solutions for big data ana- lytics systems	41
2.3	Feature overview of provenance models for nested data . . .	41
2.4	Example input data with highlighted explanations	43
2.5	Query-based explanations for the missing example output based on reparameterizations	48
2.6	Possible instance-based explanation for the missing example output	49
3.1	Evaluation semantics and output types for the operators of our NRAB.	60

4.1	Unexpected data that is existing in the example output	76
5.1	Shortened example input Relation R_{short}	107
5.2	Output of $\llbracket F_{address1}^I(R_{short}) \rrbracket^{SP}$	108
5.3	Operator-dependent lightweight instance extension \mathcal{LSP}^I . .	111
5.4	Unexpected data that is existing in the example output	115
5.5	Explanations for the existing example output	126
6.1	Expected data that is missing in the example output	134
6.2	Valid parameter changes	137
6.3	Result of $Q_{example}$ and the reparameterized query Q_0	141
6.4	Result of the reparameterized query Q_2	141
6.5	Result of the reparameterized query Q_1	141
6.6	Result of the reparameterized query Q_3	141
6.7	Shortened input data R_{short} from Table 1.1 after annotation .	153
6.8	Output R_F of $\llbracket F_{address1}^I(R_{short}) \rrbracket^{SP}$	156
6.9	Output of $\llbracket \sigma_{(R_F)} \rrbracket^{SP}$	159
6.10	Intermediate relation nesting results	163
6.11	Detailed captured annotations stored in attribute p_S1_4 of tuple 91 in Table 6.10	163
6.12	Relation nesting result R_{NR} after merging the individual re- sults in Table 6.10	164
6.13	<i>retained</i> tuples in $R_{S_2}^A$ after selection, projection, and dedu- plication	170
8.1	Cluster setups of the two Spark clusters used for the evaluation	197
8.2	Evaluation scenarios for the tree-pattern matching algorithm	200
8.3	Number of operators in the query pipeline	205
8.4	Evaluation scenarios for the Pebble algorithm	207
8.5	Evaluation scenarios for the Breadcrumb algorithm	221
8.6	Summary of explanations returned for the lineage-based approach WN++, our reparameterization-based approach without SAs (RPnoSA) and our fully fledged approach RP . .	230

LIST OF DEFINITIONS

3.1	Nested Relation Schema	52
3.2	Nested Relation Instance	52
3.3	Label function	53
3.4	Tuple projection	54
3.5	Tuple concatenation	54
3.6	Multiplicities	54
3.7	Path	55
3.8	Query execution	58
4.1	Value constraint	74
4.2	Cardinality constraint	75
4.3	Tree-pattern node	75
4.4	Structural constraint	75
4.5	Tree-pattern edge	76
4.6	Tree-pattern	76
4.7	Schema-matching	78
4.8	Data-matching	79
5.1	Access provenance	95
5.2	Input provenance	95
5.3	Manipulation provenance	96
5.4	Strucural provenance capture extension	97

5.5	Structural provenance execution	97
5.6	Lightweight schema extension	109
5.7	Lightweight instance extension	110
5.8	Lightweight structural provenance	111
5.9	Provenance tree	114
5.10	Backtracing structure	114
6.1	Nested instances with placeholders	133
6.2	Matching NIPs	133
6.3	Why-not question	134
6.4	Valid parameter changes	136
6.5	Reparameterizations (RPs)	137
6.6	Successful Reparameterizations (SRs)	137
6.7	Partial order over SRs \preceq_{ϕ}	138
6.8	Minimal successful reparameterizations (MSRs)	139
6.9	Explanations	139