Debugging Missing Answers for Spark Queries over Nested Data with Breadcrumb

Ralf Diestelkämper
University of Stuttgart - IPVS, Germany
ralf.diestelkaemper@ipvs.uni-stuttgart.de

Seokki Lee
University of Cincinnati, USA
lee5sk@ucmail.uc.edu

Boris Glavic
Illinois Institute of Technology, USA
bglavic@iit.edu

Melanie Herschel
University of Stuttgart - IPVS, Germany
melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

We present Breadcrumb, a system that aids developers in debugging queries through query-based explanations for missing answers. Given as input a query and an expected, but missing, query result, Breadcrumb identifies operators in the input query that are responsible for the failure to derive the missing answer. These operators form explanations that guide developers who can then focus their debugging efforts on fixing these parts of the query. Breadcrumb is implemented on top of Apache Spark. Our approach is the first that scales to big data dimensions and is capable of finding explanations for common errors in queries over nested and de-normalized data, e.g., errors based on misinterpreting schema semantics.

PVLDB Reference Format:
Ralf Diestelkämper, Seokki Lee, Boris Glavic, and Melanie Herschel. Debugging Missing Answers for Spark Queries over Nested Data with Breadcrumb. PVLDB, 14(12): 2731 - 2734, 2021.
doi:10.14778/3476311.3476331

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UniStuttgart-DataEngineering/breadcrumb.

1 INTRODUCTION

Data-intensive scalable computing (DISC) systems, e.g., Apache Spark and Flink, enable scalable processing of queries over (nested) data stored in a wide variety of data formats. Like database systems, DISC systems allow developers to express their queries in a high-level declarative language that abstracts away lower-level details of data distribution, fault tolerance, and distributed execution. However, support for debugging queries for these systems is limited compared to debugging support for programming languages.

One common problem that arises in debugging is that a query fails to return an expected answer. Several types of explanations for such *missing answers* have been studied in past work (as surveyed, e.g., in [8, 9]). In this work, we focus on *query-based explanations* as originally proposed in [4]. Such explanations aid data engineers in their debugging tasks by identifying parts of the query that should be repaired to return an answer that the developer did expect to see in the result. For instance, a selection operator may be part of an explanation if the selection condition is too strict, causing the missing answer of interest to be filtered out. So far, research on query-based explanations has primarily focused on relational data and queries limited to subclasses of relational algebra plus aggregation [2–4, 6]. However, while it would be possible to implement these algorithms in a DISC system, none of these approaches addresses the following challenges stemming from the characteristics of typical DISC workloads:

Challenge 1: Nested data. Nested data formats such as JSON are common in DISC workloads. Misuse of attributes in operations such as flattening and nesting that restructure nested data are typical sources of errors for queries over nested data. Past work neither supports nested data nor detects such errors.

Challenge 2: Denormalized schemas. One advantage of DISC systems is that they allow queries to directly access raw data without the need for designing a relational schema and transforming the data into this schema. The net result is that datasets processed by such systems are often denormalized and have several hundreds of attributes. Furthermore, data stored in a data lake is typically not sufficiently documented. Thus, developers often have to make educated guesses about the semantics of attributes, leading to errors when wrong attributes are used in a query. Past work on query-based explanations does not account for this type of errors, e.g., projection operators are not considered as sources of errors.

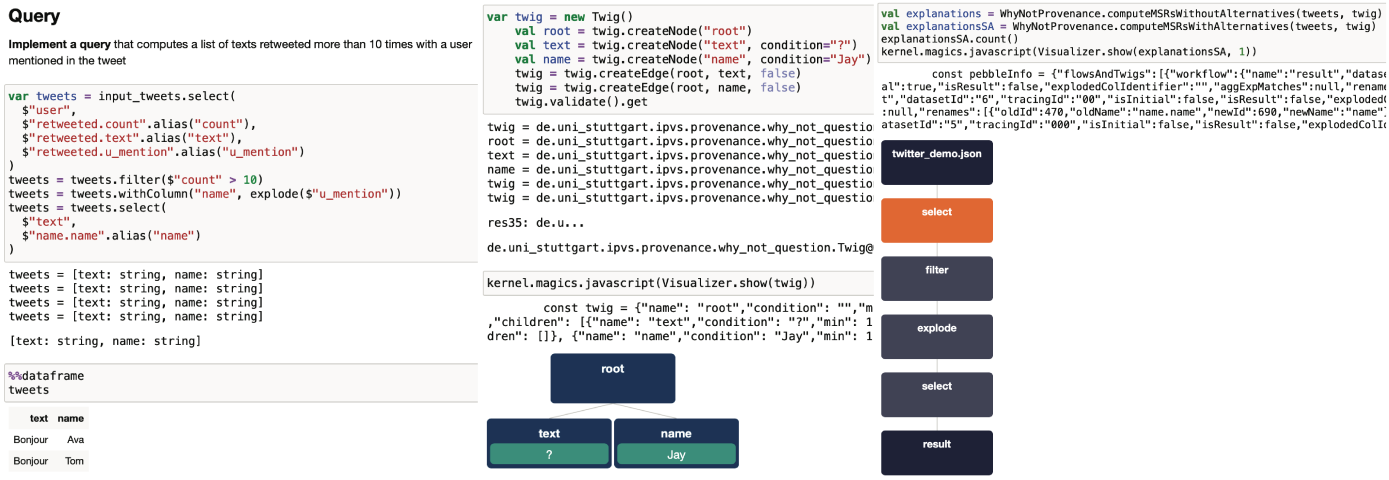
Challenge 3: Scalability. Typical DISC workloads process 100s of GBs of data. However, existing solutions for query-based explanations typically only scale to datasets that are a few MBs in size. The main reason for this lack of scalability is that these methods rely on tracing full provenance (e.g., [5]) and have to check intermediate results produced by query operators to determine when tuples that could have produced the missing answer "got lost".

In this paper, we present *Breadcrumb*, a system for explaining missing answers in Apache Spark that addresses these challenges. To the best of our knowledge, Breadcrumb is the first system capable of producing explanations for missing answers over (nested) datasets of realistic scale (100s of GBs). The system is built upon the scientific contributions detailed in [7][1], which we briefly summarize in the following. Breadcrumb is available on GitHub[2].

Schema alternatives. Breadcrumb considers attributes of the same type and structure from the input schema as potential alternatives to

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476331

[1] An extended version is available at https://arxiv.org/abs/2103.07561
[2] https://github.com/UniStuttgart-DataEngineering/breadcrumb



(a) (b) (c)

Figure 1: Breadcrumb in action: For a given query, query result (a), and an expected but missing answer specified as a tree pattern (b), Breadcrumb computes and visualizes query-based explanations for missing answers matching the tree pattern (c).

each other. For example, a developer may have accidentally referred to a home address instead of a work address attribute in their query. Through reasoning about schema alternatives, Breadcrumb can identify and explain errors caused by misinterpretation of attribute semantics in operators such as projection, flattening, and nesting.

Re-parameterizations. To support nested data, we developed an approach for generating explanations for missing answers over queries expressed in a nested relational algebra for bags. In previous work, the lineage [8] of the input query has been used to identify selection and join operators that filter rows (called *compatibles*) which could have contributed to the derivation of the missing data of interest. Such selections and joins are causes of the missing answer, because their conditions could be modified to not filter compatibles, thus repairing the query to return the missing answer. However, lineage is not sufficient for identifying operators that are responsible for the failure to derive an answer, because it does not provide any information about whether replacing a reference to an attribute with one of its alternatives (e.g., replacing work with home address) can repair the query to return the missing answer. To address this shortcoming, we developed a novel formalization of explanations for missing data based on *re-parameterizations* which are repairs of the input query that modify operator parameters. Defining explanations in this way, we can guarantee that for each explanation (set of operators) there exists at least one repair of the query that returns the missing answer and modifies precisely the operators from the explanation.

Efficiency and scalability. Computing explanations based on our new formalism is NP-hard in data complexity. However, under some reasonable restrictions, the problem is in PTIME [7]. Nonetheless, even the restricted version is still too expensive. Instead, Breadcrumb computes explanations heuristically in a two-step process. First, it evaluates a single instrumented version of the input query that annotates each result tuple with several boolean flags that indicate under which schema alternatives the tuple may be in the result, which selection operators would have filtered the tuple (under each schema alternative), and more. This instrumentation rewrites the original query plan to return a superset of the original query result with each row annotated with these boolean flags. In a second step, explanations are extracted from the final annotated query result. In contrast to prior work, Breadcrumb avoids materializing fine-grained provenance information and does not require analysis of intermediate query results. Furthermore, we have taken care to avoid operations such as cross products that do not scale. While the approach is heuristic, we have demonstrated [7] that Breadcrumb typically finds a superset of the explanations produced by approaches from past work.

Figure 1 shows screenshots of Breadcrumb integrated in a Jupyter notebook and using Apache Spark as a backend. The screenshots show the three main steps of using the system to explain missing answers: (a) the user identifies a missing answer of interest based on a given query result; (b) the user specifies the missing answers to be explained as a tree pattern (the user can leave some parts of the missing answer unspecified); and (c) the system computes explanations that the user can then explore using Breadcrumb's interface. Note that while our system is implemented on top of Spark, many of the techniques developed for Breadcrumb are of independent interest and apply to other use cases, e.g., relational databases, as well. We discuss the individual steps of using Breadcrumb in more detail next (see our accompanying video: https://youtu.be/Y0uWqdtWGGw).

2 EXAMPLE SCENARIO

We explain Breadcrumb's capabilities using an example session of a user interacting with our system's Jupyter UI (https://jupyter.org/) as shown in Figure 1. Consider the simplified Twitter dataset shown in Table 1. Each tweet consists of a `user`, `retweeted` tweets (that the user re-posted without further comments), and `quoted` tweets (that the user retweeted with additional comments). The example Spark query shown in Figure 1a returns texts of tweets that are retweeted more than 10 times paired with a user mentioned in the retweeted tweet. The corresponding operator pipeline is shown in Figure 2. It first flattens the retweeted tweets ($\texttt{flatten}_T$), then filters tweets based on the condition `count > 10` ($\texttt{selection}_{count>10}$), next

user	retweeted			quoted		
	count	text	u_mention			
Peter	1	Hello	name: Jay			
Lisa				count	text	u_mention
				42	Moin	name: Star, Jay
Jim	count	text	u_mention	count	text	u_mention
	13	Bonjour	name: Ava, Tom	27	Hola	name: Star, Mia

Table 1: Example input data: simplified tweets

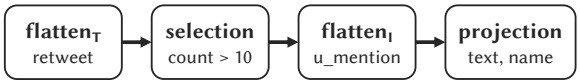


Figure 2: Example query

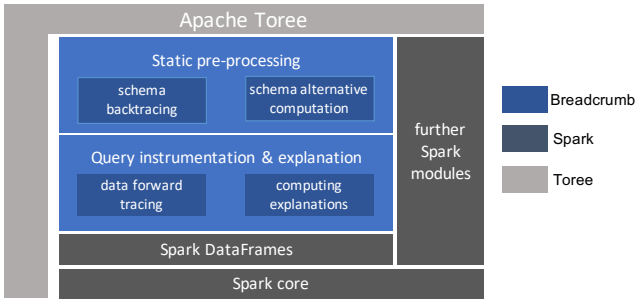


Figure 3: Breadcrumb's architecture

flattens user mentions of the remaining tweets ($\texttt{flatten}_I$), and finally projects on `text` and the `name` of each user mentioned in the retweet ($\texttt{projection}_{text,name}$).

Breadcrumb programs are specified using Spark's DataFrame API. Formally, our approach is based on a nested algebra for bags similar to [11]. Breadcrumb supports projection, selection, renaming, equi-join and outer-joins, flattening of nested tuples ($\texttt{flatten}_T$) and relations, tuple and relation nesting, aggregation, union, and deduplication. This set of operators is significantly larger than the set of operators supported by existing approaches.

Returning to our example, the result of the program over the example data is shown at the bottom of Figure 1a. Assume that the developer notices that Jay, a user expected to be in the result, is not returned by the query and uses Breadcrumb to explain this missing answer. To initiate this investigation, the developer has to provide Breadcrumb with the input data and program (shown before) and a why-not question that expresses what missing answers to focus on. To this end, Breadcrumb leverages tree-patterns [12]. These patterns define the nesting structure of the missing answer as well as permissible values. Placeholders can be used to leave some values unspecified, i.e., a placeholder can stand in for any value. The example question can be expressed as the tree pattern shown at the bottom of Figure 1b. The developer expects an answer with attribute `name` bound to value Jay and with any value for attribute `text` (placeholder ?). The top part of Figure 1b shows how a tree pattern is created in Breadcrumb by creating node and edge objects in Scala. Nested relations in a tree pattern may also contain the placeholder $*$ that represents any number of tuples. Furthermore, tree patterns may contain ancestor-decendant relationships in addition to direct edges (the edge's boolean flag is set to true).

Having created these inputs, the developer can call Breadcrumb to compute explanations. Intuitively, each explanation encodes a set of operators from the query such that there exists a **re-parameterization** which produces at least one answer matching the tree pattern (the missing answer). Recall that a re-parameterization is a repair that changes parameters of all operators in an explanation, but preserves the parameters of the other operators and the query structure. When computing explanations for missing answers, Breadcrumb considers schema alternatives for re-parameterizations to be able to explain errors caused by misinterpretation of attribute semantics. To efficiently compute explanations, Breadcrumb rewrites the original query plan to obtain an annotated query result. The annotated result contains all information necessary for computing explanations for each possible schema alternative. In Section 3, we describe this process in more detail.

Breadcrumb returns two explanations for the running example. The first one only consists of the filter operator ($\texttt{selection}_{count>10}$ in Figure 2). The filter prevents Jay from appearing in the result since the retweet mentioning Jay (part of the first nested tuple in Table 1) only has a retweet count of 1. Replacing the constant 10 with, e.g., 0, in the filter operator causes the tuple (Hello,Jay) matching the tree pattern to be returned. Note that this explanation does not use a schema alternative, since the fix does not change an attribute reference. The second explanation contains the first flatten operator, i.e., $\texttt{flatten}_T$ in Figure 2. This operator is a potential cause of the missing answer under the aforementioned schema alternative. Recall that this schema alternative replaces the `retweeted` with the `quoted` attribute in the parameters of operator $\texttt{flatten}_T$. It corresponds to the assumption that the developer may actually be interested in quoted instead of retweeted tweets. When applying a re-parameterization corresponding to this explanation, the developer obtains the tweet (Moin,Jay) in the result. For demonstration purposes, the system provides two implementations to compute explanations: one that ignores schema alternatives and one that uses schema alternatives. Figure 1c shows the two corresponding code snippets (the function call which does not consider schema alternatives is shown on top). Currently, Breadcrumb requires the schema alternatives for the input schema to be manually provided as input to the system, e.g., the user has to specify that the quoted attribute and retweeted attribute are alternatives for each other. Integrating schema matching techniques or schema-free query processors [1, 10] to automatically identify alternatives is an interesting avenue for future work. Also note that the visualization of the result uses labels for nodes in the operator pipeline that differ from the (internal) operator names to make it easier for users to match the operator with a line of code in their query.

3 THE BREADCRUMB ARCHITECTURE

Breadcrumb extends Apache Spark with the means to compute explanations. As depicted in Figure 3, Breadcrumb's modules (in blue) are divided into two categories that correspond to the two

main steps mentioned in Section 1: (a) modules that are data independent and pre-process the user-provided input, and (b) modules that leverage the result of pre-processing for the efficient computation of explanations. These modules extend Spark's DataFrame API and rewrite query plans obtained from Spark's query planner Catalyst (dark grey). In this demo, we use Jupyter notebooks with the Apache Toree kernel as an interactive frontend for Breadcrumb (light grey).

Recall that an explanation is a set of operators such that there exists a repair of the query that modifies precisely these operators and returns an answer that matches the why-not question. We call such repairs successful re-parameterizations. Ideally, the repairs corresponding to an explanation should be minimal, i.e., they should not unnecessarily modify operators or result in unnecessary side-effects (changes to the query result). We call such repairs *minimal successful re-parameterizations* (*MSRs*). While our formalization requires explanations to correspond to MSRs, the approach implemented in Breadcrumb is heuristic and cannot guarantee minimality in all cases. We now briefly describe each of Breadcrumb's components involved in the process of generating explanations.

Pre-processing. Given the query and why-not question provided via the user interface as well as schema alternatives, Breadcrumb first calls the modules for static pre-processing. These modules do not access the input data. **Schema backtracing** (step 1) determines selection conditions over the input table by back-propagating constraints defined in the why-not questions to attributes in the input schema. For every query repair, the inferred selection conditions have to hold for all input tuples that may be involved in the derivation of the missing answer. Later, these conditions are applied to prune data that is irrelevant for computing explanations. Schema backtracing resembles the approach from [3]. During the **schema alternative computation step** (step 2), Breadcrumb determines statically how to substitute attributes in the query with alternatives and enumerates all possible combinations of such substitutions. It checks for each combination whether the corresponding combined attribute substitution yields an executable query.

Query instrumentation and explanation. The two modules accessing and processing the input data are then used as follows. Given the constraints, schema alternatives derived during preprocessing, and the query, **forward tracing** (step 3) instruments the query to trace input data matching these constraints through the query's operators. For that purpose, Breadcrumb propagates four types of boolean provenance annotations through the query operators for each schema alternative. It extends the operator semantics to propagate the annotations, compute the operator's output under all schema alternatives simultaneously, and retain tuples that would be removed by selective operators as explained above. These operator extensions are carefully designed to keep computational overhead reasonable. Upon evaluation of the instrumented query over the input data, Breadcrumb returns a query result with sufficient information to compute explanations. During the last step, **computing explanations** (step 4), Breadcrumb extracts successful re-parameterizations based on the annotated output produced by the previous step. To identify explanations that correspond to MSRs, it computes a lower and upper bound for the size of side-effects caused by any MSR for a set of operators. An exact measure of side-effects is computationally prohibitive because it would require (a) precisely knowing the side-effects caused by each possible re-parameterization involving an explanation's operators and (b) an efficient approach to compare the number of side-effects for each such re-parameterization.

Overall, our heuristic approach trades accuracy for performance and, thus, may return explanations that are not minimal and may miss explanations. See [7] for a discussion under which circumstances Breadcrumb returns accurate explanations. Before returning the explanations corresponding to MSRs, Breadcrumb orders them by the number of operators that need to be modified by re-parameterizations corresponding to an explanation. If two explanations have the same number of operators, Breadcrumb ranks the one higher that has a lower upper bound for the side-effects.

4 DEMONSTRATION EXPERIENCE

We showcase Breadcrumb through interactive debugging sessions running in a Jupyter Notebook. We will use two real world datasets (Twitter and DBLP data) as well as TPCH datasets of up to 100GB in size deployed on a cluster with six compute nodes. For each dataset, we have prepared multiple scenarios (including a scenario similar to our running example), each comprising a query, a tree pattern expressing a why-not question, and the schema alternatives for the dataset. After showcasing a (simple) scenario to familiarize attendees with the use of the system, we offer attendees the possibility to experience additional pre-cooked scenarios and/or to write their own scenarios to explore Breadcrumb's capabilities described in Section 2, e.g., in terms of expressing tree patterns, queries, or assessing scalability and explanation quality.

ACKNOWLEDGMENTS

Partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2120/1 - 390831618 and by NSF grants IIS-1956123.

REFERENCES

[1] D. Aumueller, H. Do, S. Massmann, and E. Rahm. 2005. Schema and Ontology Matching with COMA++. In *ACM Conference on the Management of Data (SIGMOD)*. 906–908.

[2] N. Bidoit, M. Herschel, and A. Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *Conference on Information and Knowledge Management (CIKM)*. 713–722.

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *Conference on Extending Database Technology (EDBT)*. 145–156.

[4] A. Chapman and H. V. Jagadish. 2009. Why not?. In *ACM Conference on the Management of Data (SIGMOD)*. 523–534.

[5] Y. Cui and J. Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal* 12, 1 (2003), 41–58.

[6] D. Deutch, N. Frost, A. Gilad, and T. Haimovich. 2018. NLProveNAns: Natural Language Provenance for Non-Answers. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 1986–1989.

[7] R. Diestelkämper, S. Lee, M. Herschel, and B. Glavic. 2021. 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)*. 405–417.

[8] B. Glavic. 2021. Data Provenance - Origins, Applications, Algorithms, and Models. *Foundations and Trends in Databases* 9, 3-4 (2021), 209–441.

[9] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.

[10] Y. Li, C. Yu, and H. V. Jagadish. 2008. Enabling Schema-Free XQuery with meaningful query focus. *The VLDB Journal* 17, 3 (2008), 355–377.

[11] L. Libkin and L. Wong. 1997. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences (JCSS)* 55, 2 (Oct. 1997), 241–272.

[12] J. Lu, T.W. Ling, Z. Bao, and C. Wang. 2011. Extended XML Tree Pattern Matching: Theories and Algorithms. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 3 (2011), 402–416.