WhyFlow: Explaining Errors in Data Flows Interactively

Maeda F. Hanafi, Azza Abouzied
New York University Abu Dhabi
Abu Dhabi, UAE
maeda.hanafi,azza@nyu.edu

Yunyao Li, Marina Danilevsky
IBM-Almaden
San Jose, CA, USA
yunyaoli,mdanile@us.ibm.com

ABSTRACT

In real-world data processing pipelines or data flows, data are often processed by many modules, potentially written by multiple developers. As bugs are introduced, it may be relatively easy to identify errors in the final outputs. However, tracing the errors backwards to their origin in the pipeline in order to identify and fix the relevant faulty module is a difficult and time-consuming task in practice. We built WhyFlow to help users localize which modules in the pipeline are propagating errors by automatically generating explanations: simple predicates that zero in on the most relevant aspects of the errors within the data flows. We describe WhyFlow and present a user study that shows how explanations are useful in helping users identify errors in data flows.

KEYWORDS

explanations, data flow, queries, interactive debugging, visualization, data pipelines, human-in-the-loop tools

1 INTRODUCTION

Nowadays, large and complex data processing data flows that integrate, extract, and transform data from multiple sources are commonplace [1, 7, 21]. These data flows engage different user groups: multiple developers contribute and update different modules within a single data flow, systems administrators deploy the data flow, and data analysts and decision makers utilize the outputs of the data flow once deployed in a production or serving environment. During serving, it is the data analyst who often detects the presence of errors through suspicious outputs and kickstarts the arduous and time-consuming task of debugging. Correctly routing the bug with sufficient information to the appropriate development team is essential to its timely handling. These end-users, however, are not familiar with each module's code-base [9] and tracing the error backwards to its origin is often difficult. End-users also often struggle to conceive possible explanations for the failure, even after suspecting a certain module to be the cause [9]. Furthermore, decision makers and data analysts often prioritize other high-value external goals over software reliability [9] and may not spend the time to accurately determine and explain the error.

Consequently, designing a tool that can help end-users localize and explain faults from a few error examples without requiring module introspection is essential, albeit challenging. Motivated by recent research in the area of statistical debugging [11], as well as in explaining errors and outliers in relational queries [19], machine learning models [6] and data sources [18], we built and evaluated an interactive tool, WhyFlow, that supports data flow debugging via labeling some of its incorrect outputs.

DaSH@KDD'20, August 24, 2020, Virtual Conference
2020.

Given fine-grained provenance of a data flow and a set of incorrect outputs, WhyFlow synthesizes *explanations* at each module: predicates that *best* capture flows with incorrect output. The user can then examine these explanations to determine which one best describes the faulty behavior and, subsequently, which module is the most likely source of error. We created WhyFlow adhering to the following design principles:

(1) *Minimal Tuning & Code Introspection.* An end-user, who may be agnostic to the underlying code operations of a data flow, should be able to label a sample of the output datapoints and accurately diagnose the errors propagating from upstream.
(2) *Surfacing Features of Incorrect Data Flows in a Visually Perceptible Fashion.* Data flow and provenance visualizations should highlight characteristic differences between the incorrect data flows and the residual data flows, both locally at the code module level, and globally at the data flow level.
(3) *Concise, Sensitive and Specific Error Explanations.* Explanations of the incorrect data flow should be simple and intuitive for an end-user to understand yet expressive enough. Ultimately, explanations serve two purposes: (1) *communication* or helping users provide a clear and correct description of an error's behavior and (2) *diagnosis* or helping users identify the source and cause of an error.

1.1 Motivating Use Case

We motivate our work by describing a typical data flow and potential error scenarios from a large retail chain with multiple stores across the US. Upper management might be interested in identifying stores with unusual customer behavior such as *unusually high rates of purchase returns.* The development team constructs a data flow that (1) integrates individual sale and customer records from multiple stores, (2) transforms and cleans the data through a complex sequence of modules, and (3) produces the final dashboards and reports for the management team.

End-users, or store managers, can sanity check these reports. Imagine a manager of such a flagged store — one with a high count of problematic customers — who believes that there actually should be no unusually high purchase return patterns. She notices several consistent, regular customers who should not belong in a list of problematic customers and passes this information back to the data flow development team.

At this point, an analysis of the entire data flow is required to track down the spurious problematic customer entries. Any of the following errors (or others) could have occurred:

(1) *Data scan errors:* A source table that tracks customer purchase returns had data entry errors with 100x higher number of returns for some customers.

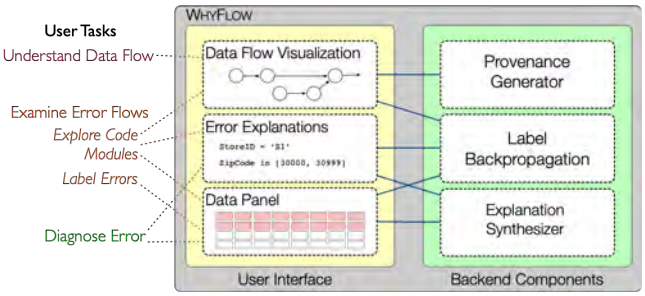

Figure 1: The interface and backend components of WhyFlow and the various user tasks they support.

(2) *Filter Errors:* Certain purchases, stores or customer groups should have been excluded but filter errors such as using '≥' instead of '>', or incorrectly set filter thresholds.
(3) *Join Errors:* Joining tables on wrong fields or using outer instead of inner joins resulted in including customers that did not even shop at a given store showing up in that store's customer returns report.

WhyFlow enables the end-users, e.g. the store managers, to visually examine the data flows and the provenance. It also synthesizes error explanations at different modules from incorrect output labels. These explanations, combined with the end-users' domain expertise can help the debugger more accurately identify the cause of the error in the data flow, allowing developers to fix the actual source of the error.

The main contributions of this work are: (1) WhyFlow, an interactive tool tailored for analyzing errors in data flows (Figure 2), (2) a user study demonstrating how explanations are useful in helping users identify the source of errors in data flows, and (3) a scalable method for generating, clustering, and ranking explanations.

1.2 Overview

In the course of debugging a data flow with WhyFlow, an end-user performs three primary tasks: (1) understanding the data flow; (2) examining error flows by (a) labeling erroneous outputs and then (b) analyzing how code modules within the data flow contribute to error flows; and (3) diagnosing errors with the help of explanations. Figure 1 illustrates how WhyFlow supports each of these tasks through its interface and backend components. In this paper, we describe WhyFlow and present a user study of WhyFlow showing how explanations are useful in debugging errors in data flows.

2 WHYFLOW INTERFACE AND LABELING

2.1 Data Flow Visualization

A ***data flow*** is an acyclic graph, where each node takes in sets of tuples from one of more source nodes and outputs a set of tuples. Figure 2 visualizes each data flow using the common workflow graph metaphor: nodes represent modules and edges represent datapoints that flow from one module to another.

WhyFlow computes and stores offline the fine-grained **why-provenance** [4] for each data flow using strategies similar to Perm [8]. To ensure interactive performance, WhyFlow randomly samples the final outputs and only presents the provenance data that produce these outputs. If a handful (10s to 100s) of labels is given, we balance the labeled and unlabeled datapoints by under-sampling the unlabeled datapoints.

2.2 Error Labeling

Users are required to label incorrect final data points, which are presented in the data flow visualization in red. All other datapoints are labeled as unknown. Explanations of the errors are generated to capture incorrect data points. In an initial design of WhyFlow, users were able to label both correct and incorrect. However, we relaxed our requirement to have correct output labels, as users often struggled with questions such as *'how many correct datapoints should I provide?'* or *'can I simply label everything other than the error datapoints that I selected as correct?'*.

2.2.1 Label Back-Propagation. Back-propagating error labels from the outputs of leaf modules in a data flow to upstream modules allows us to assess the likelihood of a specific module being the source of an error by evaluating its immediate outputs. We label each intermediate datapoint by how much it contributes to the final errors. We back-propagate a numeric value or the *degree of error* of each datapoint. Given an intermediate datapoint p with descendant datapoints D_p in the provenance graph, its degree of error ($E(p)$) is simply the mean *degree of error* of its descendant datapoints ($d \in D_p$). The degree of error of a final output (a terminal datapoint) is 1 if the user labeled it as an error and 0 otherwise.

$$E(p) = \begin{cases} 1, & \text{if } p \text{ is an error output} \\ |D_p|^{-1} \sum_{d \in D_p} E(d), & \text{if } p \text{ is an intermediate datapoint} \\ 0, & \text{otherwise} \end{cases}$$

In Figure 2, each edge is color coded by each datapoint's degree of error: light shades of red for low degree of error and deeper shades for higher degree of error; gray for datapoints whose contribution to final errors is unknown. This allows end-users to quickly eliminate branches that do not contribute to any error.

2.3 The Data Panel

The data panel displays a selected module's output datapoints in tabular form. For each column, a simple profiler determines from the provenance data whether a module (i) *added* a new column (⊙), (ii) *modified* the values of a column (Δ), (iii) *maintained* the values of a column, or (iv) used the column as a *key* to join multiple inputs (⚷). Each column in the data panel is annotated with its type icon, ⊙, Δ, ⚷, except for columns whose values remained the same. These annotations allow the users to understand at a high-level the behavior of a data flow even with black-box modules.

3 EXPLAINING ERRORS

An explanation e is a predicate as defined by the explanation language in Listing 1 that operates on the attributes of a module's output datapoints. The explanation language is concise, sensitive and specific. Given its simplicity, end-users can quickly examine and choose the appropriate explanation. To maintain efficient synthesis we restricted our conjunctions to three atomic conjuncts, and to three disjunctions of such conjuncts. We found that this size

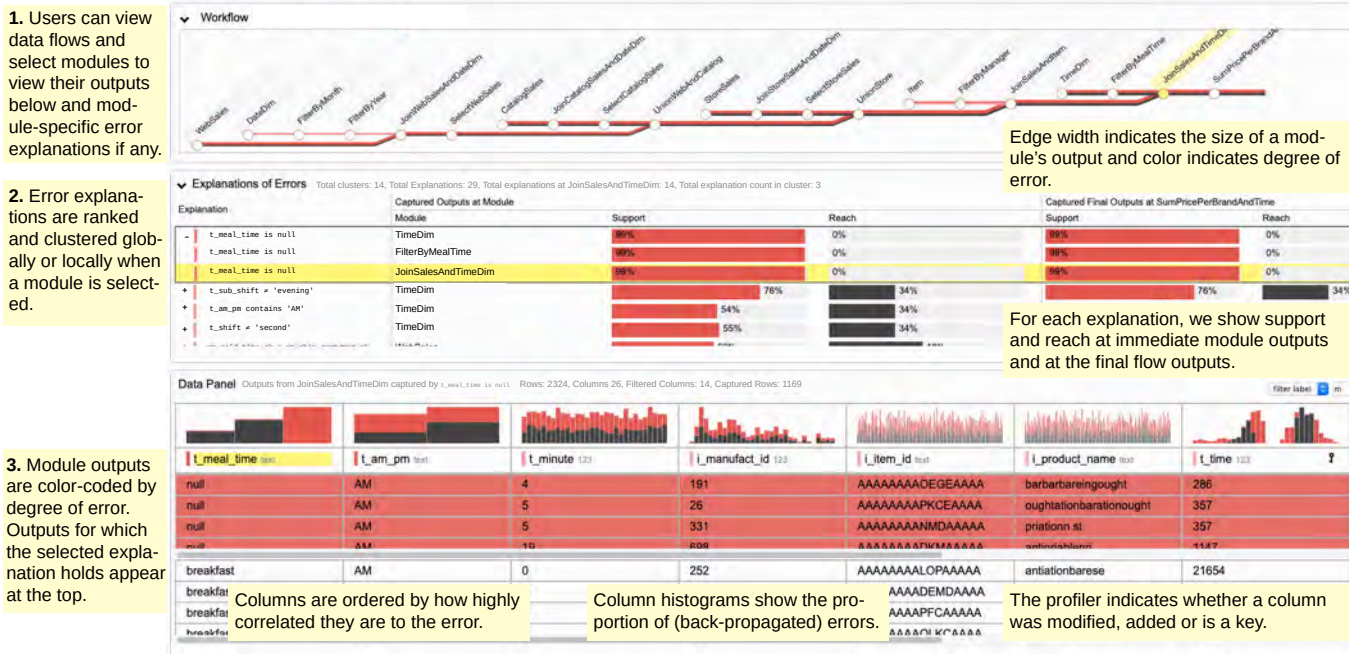

Figure 2: Interface of WhyFlow

restriction is important to ensure explanation conciseness, and thus user-readability: the maximum size of an explanation is 9 atoms — Miller's law states that the magic number seven +/- two is a rough limit on our cognitive capacity to process information [14]. It also allows for interactive synthesis from a few labeled outputs. Our explanations capture filter and join errors, which are a major class of data flow errors. However, they cannot explain errors affecting modules that *reduce* a set of inputs. To explain such aggregation errors, we need to introduce quantification, which may cause the synthesis problem to become intractable.

For each module in the flow with incorrect outputs, the synthesizer generates explanations with *support* and *reach* within certain thresholds (see appendix for synthesis details):

Support: The percentage of *incorrect* data points captured by an explanation. For an explanation at a given module, we can evaluate support on its immediate outputs (using back-propagated labels or degree of error) or on the data flow's final outputs. We compute and present both support measures to the user in WhyFlow (Figure 2).

Reach: The percentage of *unknown* (unlabeled) data points captured by an explanation. For an explanation at a given module, we can evaluate reach on its immediate outputs (outputs with a degree of error equal to zero) or on the data flow's final outputs (i.e. unlabeled outputs, which may or may not correct). We compute and present both reach measures to the user in WhyFlow.

3.1 Global Explanation Clustering & Ranking

To minimize the user's cognitive load of exploring many explanations, we cluster explanations by *semantic similarity*. For example, the explanations `size < 6` and `size in [0, 6]` are semantically identical for size, a non-negative integer, but both are generated; WhyFlow should group these explanations together. To approximate the notion of semantic similarity without statically analyzing each explanation, we cluster explanations with identical column sets. Consider explanations e_1, e_2, e_3 that operate over columns {A, B, C}, {A, B}, and {A, B} respectively: explanation e_1 will appear in one cluster and explanations e_2, e_3 will appear in another cluster.

To rank explanations and clusters, we assign a score to each explanation e_m at a given module m. Let $C(e_m)$ be an explanation's column set. The score is a weighted sum of the following factors:

(1) An explanation's *support* (support(e_m)), *reach* (reach(e_m)), and *conciseness* (size$(e_m)^{-1}$) — smaller, minimal explanations are preferred as being easier to read and understand.
(2) The module's *maximum depth* (MaxDepth$(m)^{-1}$) within a data flow— given two otherwise identical explanations operating on different modules, the explanation from an upstream module is preferable to a downstream one as we would like to localize errors to the earliest possible origin.
(3) The *mean poorness-of-fit* across all columns in $C(e_m)$. For $c \in C(e_m)$, $F(c)$ is the *poorness of fit* of the distribution of values with a non-zero degree of error to the distribution of values with unknown degree of error as measured by either (i) 1 − the p-value of a Chi-Square test or (ii) 1 − the p-value of a Kolmogorov-Smirnov (KS) test. We use Chi-Square for categorical values. Explanations over columns that show more differences between the distribution of values for errors and unknowns are preferred as they allow users to visually verify the contribution of a column to an error.
(4) The *mean relationship* (M) between the module and all columns of an explanation's column set (Section 2.3) — explanations on columns that are modified, added or used as candidate join keys by the given module are more relevant.

```
Proposition p::=a contains (str | b) |
                a starts with (str | b) |
                a ends with (str | b) |
                a matches regex |
                a = (x | str | b | NULL) |
                a ( > | >= | < | <= ) (x | b) |
                a in [min, max]
Atom t       ::= p | not(p) | true
Conjunction c::= t and t and t | false
Explanation e::= c or c or c
```

Listing 1: WhyFlow's language of explanations. `str` is a string constant; `x`, `min`, and `max` are numeric constants, `regex` is a regular expression, and `a`, `b` denote attributes

The overall score of an explanation is:

$$\text{Score(e)} = \alpha_1 \cdot \text{support}(e_m) + \alpha_2 \cdot (1 - \text{reach}(e_m)) + \alpha_3 \cdot \text{size}(e_m)^{-1} + \alpha_4 \cdot \text{MaxDepth}(m)^{-1} + \alpha_5 \cdot \frac{1}{|C(e_m)|} \sum_{c \in C(e_m)} F(c) + \alpha_6 \cdot \frac{1}{|C(e_m)|} \sum_{c \in C(e_m)} M(c)$$

Within each cluster, explanations are ordered by highest scoring explanation first. A cluster's score is set to be the score of the highest scoring explanation within it. Globally, all clusters are sorted in descending order of score (see Figure 2).

4 EVALUATION

4.1 End-to-End User Evaluation

We conducted a comparative user study of debugging with WhyFlow and a baseline version of WhyFlow, where explanation suggestions, and workflow edge color coding by error were disabled. The Baseline version mimics manual debugging, limiting the workflow visualization to displaying, upon clicking a module, the outputs in a tabular format. To isolate the effect of explanations on end-user debugging, we perfectly labeled intermediate data points as correct or incorrect. Additionally, in the study, explanations were ranked by only support and reach. Later, we evaluate the effectiveness of global ranking separately.

Data Flows. We reproduced two data flows from the TPC-Decision Support (TPC-DS) benchmark queries and data set [16]. The TPC-DS benchmark is widely used in systems research and industry to evaluate general purpose decision support systems and big data systems. Table 1 describes these two data flows.

The modules in our data flows can generally be described as a single or simple sequence of relational operations such as 'filter' or 'join and aggregate.' While data flows in WhyFlow can have non-relational modules, the modules were relational to simplify the debugging task for users not familiar with the data flow. We label each black-box module with such relational semantics, which are surfaced to participants to enable them to more easily grasp the gist of a data flow without necessarily understanding the implementation-level details. We then purposefully introduce relational operation errors into particular modules.

Participants and Methods. We recruited 10 participants who have taken an introductory database applications course. Participants

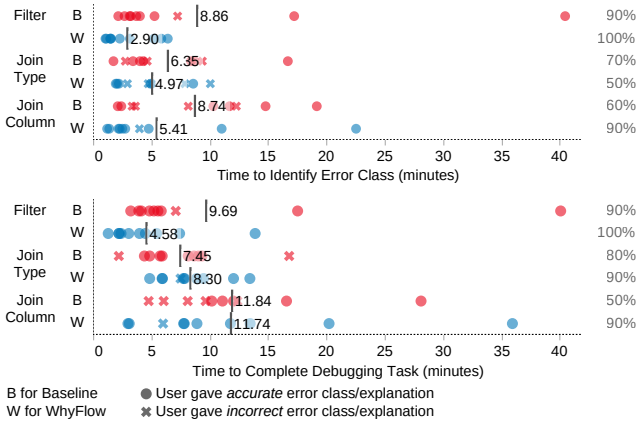

Figure 3: The duration of debugging activities for each user. The avg time per error class and tool are also marked.

were familiar with the standard relational operators, but not with how they were specifically implemented in the data flows. Each participant was given a training session for both conditions of WhyFlow and mock debugging tasks on a mock data flow to get familiar with the tool conditions and the scope of the code modules and errors they would analyze.

We presented the participants with correct executions of data flows W1 and W2 in WhyFlow and asked them to describe each code module's expected operation and identify the columns it operated on. Each participant then completed three debugging tasks for different error classes on each condition of WhyFlow. We randomized the order of conditions they started with and the order of tasks within each condition:

(1) **Filter Errors:** Modify a filter operation such that it selects correct and incorrect values in the output.
(2) **Join Type Errors:** Modify a join type replacing inner joins with full outer joins.
(3) **Join Column Errors:** Modify the parameters of the join operation causing the join to operate on incorrect columns.

For each error class, we created two similar complexity variants (Table 2) to minimize learning effects across Baseline and WhyFlow. These three error classes cover the types of errors WhyFlow can explain. We pre-labeled the correct and incorrect outputs. To complete a debugging task, participants had to either select an explanation for the error in WhyFlow or provide one for Baseline. Participants were asked to think aloud and verbalize their thoughts when they believed they had identified the faulty module, determined the error class, or could express some intuition on the nature of the error. These events, and all other participant verbalizations during the course of the experiment, were recorded.

4.1.1 Explanations Matter. We hypothesized that users would identify errors and select appropriate explanations more accurately using WhyFlow vs using Baseline without requiring more time.

We found that ***users selected correct explanations more frequently with*** WhyFlow ***than with Baseline*** (Table 3). Remarkably, all five users who initially misidentified the join type error class with WhyFlow selected the correct explanation.

Data Flow	Data Set	Description	Modules	Sources
W1	TPC-DS	Which customers shopped at which stores and returned more than 1.2 of the average store's return amount?	9	4
W2	TPC-DS	What are the sums of sales prices per customer location?	10	4
W3	TPC-DS	Given a time period and a particular manager, what brands were sold, when (hour and minute of the day) were those brands sold, and what was the brand's total sales price?	15	6
W4	TPC-H	Given a set of countries and certain requests, how many customers per country had those requests and what is the total account balance per country?	5	2
W5	TPC-H	For each item's brand, its type, and its size, how many available suppliers are there?	9	2
W6	TPC-H	How many items per container type are still in the production line or are shipped out?	9	2

Table 1: Data Flow descriptions. The first two data flows were used for the user study (Section 4).

ID	Expected Operation	Error	Sample Valid, Relevant Explanation
W1-JStore	Inner Join on W1.JoinWithStore	Outer Join	`storeID = NULL`
W1-JCust	Inner Join on W1.JoinWithCustomer	Outer Join	`customerID = NULL`
W1-CStore	Join on store IDs	Join on store ID and company ID	`not(Returns.storeID = Store.storeID)`
W1-CCust	Join on customer IDs	Join on customer ID and market ID	`not(Returns.customerID = Customer.customerID)`
W2-Zip	Selects zipcodes that start with '80'	Selects zipcodes that start with '80' and '30'	`zipcode starts with '30'`
W2-State	Selects states in {CA,GA,WA}	Selects states in {CA,GA,WA,CT}	`state = 'CT'`
W2-Quarter	Selects yearly quarters = 2	Selects yearly quarters ≤ 2	`quarter = 1`
W3-Manager	Selects managerID = 1	Selects managerID ≤ 2	`not(managerID = 1)`
W3-Meal	Selects meal times in {breakfast, dinner}	Selects meal times in {null,breakfast,dinner}	`mealtime = NULL`
W4-Code	Selects country codes that begin with 1	Selects any country codes	`countrycode >= 20`
W4-Comment	Selects comments not containing 'deposits' and containing 'special requests'	Selects comments containing 'deposits' or containing 'special requests'	`comment contains 'deposits' or comment contains 'special'`
W5-Type	Selects types that begin with 'SMALL ANODIZED'	Selects types that begin with 'SMALL'	`not(type contains 'ANODIZED')`
W5-Size	Selects sizes in {8,12,14,25,30,34,36, 41}	Selects sizes in {8,12,14,25,30,34,36, 41,42}	`size >= 42`
W6-Ship	Selects shipmode in {RAIL}	Selects shipmode in {RAIL, SHIP, TRUCK}	`not(shipmode = 'RAIL')`
W6-Size	Selects sizes between 5 and 26	Selects sizes < 6 or sizes > 26	`(size < 6) or (size > 26)`

Table 2: The 15 different errors scenarios used in evaluations. The first six error scenarios were in the user study (Section 4).

Error Class	Tool	Class Identified Correctly	Correct Explanation? *No*	Correct Explanation? *Yes*
Filter	Baseline	No	10%	-
		Yes	-	90%
	WhyFlow	No	-	-
		Yes	-	100%
Join Type	Baseline	No	10%	20%
		Yes	10%	60%
	WhyFlow	No	-	50%
		Yes	10%	40%
Join Column	Baseline	No	40%	-
		Yes	10%	50%
	WhyFlow	No	10%	-
		Yes	-	90%

Table 3: Pct of users broken down by correctness of verbalized error class and correctness of the final explanation.

We conducted a two-way repeated-measures ANOVA on the duration of time for users to verbalize the error class, with error class and tool used (WhyFlow vs Baseline) as independent factors. There were neither significant interaction effects, nor a significant main effect of error class. We did find a significant main effect of tool used ($F_{1,9} = 10.15, p = 0.01$). ***Users spent less time identifying the error class with*** WhyFlow**.** Furthermore, with the exception of join type errors, their initial intuition of the error was more accurate than with Baseline (See Figure 3).

We also conducted a two-way repeated-measures ANOVA on the overall time for users to complete the debugging task and found no significant interaction effects, nor any significant main effects of tool used (WhyFlow vs Baseline). In fact, any gains WhyFlow provided to help users determine the error class were lost due to the time spent searching for and selecting an appropriate explanation. These results emphasize the importance of going beyond only support and reach for ranking explanations.

ID	Explanation Rank			Cluster Rank		
	Global	S/R	# of Expls.	Global	S/R	# of Clusters
W1-JStore	10	556	1548	1	207	598
W1-JCust	8	83	207	2	77	157
W1-CStore	14	60	141	11	26	39
W1-CCust	39	33	43	20	18	21
W2-Zip	2	12	79	1	3	22
W2-Quarter	8	87	391	3	4	140
W2-State	6	6	379	1	1	88
W3-Manager	6	4	65	1	1	24
W3-Meal	3	74	186	1	2	54
W4-Code	9	8	33	5	5	13
W4-Comment	67	13	124	20	1	29
W5-Type	5	5	307	1	1	10
W5-Size	6	6	66	1	1	8
W6-Ship	8	8	26	2	2	3
W6-Size	6	10	430	2	1	17

Table 4: WhyFlow's global explanation ranking quality. (S/R) shows the rank order using only support and reach.

Overall these results indicate that WhyFlow can be an effective data flow debugging tool in that users select more accurate explanations with WhyFlow. 90% of the users found the explanations generated by WhyFlow simple and easy to understand. One user elaborated, "I had no clue what was going on [in Baseline], but explanations gave me hints." Many users stated that they used explanations as a means to validate their intuition on what was the cause of the error. One user said: "I debugged in my head first and then I selected an explanation." Another user stated "explanations helped especially when I wasn't 100% sure".

4.2 Effectiveness of Global Ranking

We now evaluate the effectiveness of WhyFlow's global ranking of explanations. Table 4 presents the rank of the most relevant explanation for all the error scenarios described in Table 2. For 12 out of the 15 error scenarios, WhyFlow's global ranking places a valid explanation cluster within its top-5 clusters and a valid explanation within its top-10 suggestions (if we expand all clusters).[1] The table also shows that clustering reduces the number of explanation-groups that users have to examine. Finally, we observe that most error scenarios benefit from using our global ranking score over rather than only using support and reach to rank explanations. In only three scenarios (W1-CStore, W1-CCust, W4-Comment), the valid explanations are ranked lower: this is because explanations over highly correlated columns, which have more distinct error distributions, were ranked higher than the real explanations.

5 RELATED WORK

Our work draws from research in data provenance, notions of causality and explanations in databases and AI, and human-centered research on data and software debugging tools.

WhyFlow assumes the existence of provenance or data lineage. Interactive provenance querying systems allow users to investigate a specific set of suspect outputs and to audit and verify the correctness of data transformations. A survey of provenance tools can be found in [4]. WhyFlow expands on this basic debugging support by allowing users to label error outputs and visually compare the color-coded error and non-error data flows in the interface.

Causality, which characterizes the relationship between an event and outcome, is of clear practical importance to debugging. Unfortunately quantifying causal contributions is both NP-complete and limited to *data debugging* [13] or finding errors in the data source rather than the data processing modules. *Explanations* or predicates that describe outliers, anomalies, or interventions generally give up the notion of causality to enable more practical debugging [17–19]. While WhyFlow does not focus on identifying and removing specific inputs that would resolve the errors [20], WhyFlow aims to prevent errors in future executions of the data flow by helping end-users pinpoint faulty code modules through differences of error and residual outputs, including intermediate outputs and not just the data sources and final outputs [2].

Data cleaning and repair tools[5] differ from WhyFlow as they do not aim to explain the source of the errors but only to identify incorrect items in a dataset. Such tools can be used as complements to automatically label erroneous outputs in WhyFlow.

Statistical debugging combines machine learning and software debugging to select a small set of program predicates that can succinctly capture failure modes and thereby localize faults, from samples of successful and failed software run-time profiles [11, 12]. However, such approaches operate at a much lower level, analyzing both the control and data flow of a program. WhyFlow only examines the provenance graph and assumes modules are black-boxes.

WhyFlow is a mixed-initiative, programming by example (PBE) tool in that the synthesis of error explanations is guided by providing a few examples of a data flow's erroneous outputs. While debugging strategies can vary widely, a number of studies show that the debugging process is a hypothesis-driven activity [10]. In particular, most developers begin with a 'why' question regarding the program's behavior, and then skillfully transform this question into low-level tasks such as searching logs or instrumenting the code with breakpoints and print statements [10], often guided by a hypothesis on the possible location of the fault [15]. In WhyFlow, users do not directly pose why-questions; however by labeling erroneous outputs, they are implicitly using our tool to determine *why* there are differences between the error and non-error flows.

6 CONCLUSIONS & FUTURE WORK

Given end-users error labels in the final outputs, WhyFlow helps end-users with debugging data flows by explaining errors with predicates and visualizing the data flow, the provenance of outputs, and the differences between erroneous and residual data flows. Our evaluations show that WhyFlow enables most users to accurately localize and communicate faults. We hope to (i) make our explanation language even more powerful to capture aggregation errors while maintaining synthesis efficiency, (ii) utilize why-not provenance [3] to explain errors where erroneous data points are completely dropped, (iii) scale WhyFlow to handle larger data sets interactively, and (iv) expand support for data flows with several errors.

[1] For this experiment, only a fraction of the error outputs were labeled.

REFERENCES

[1] Sreeram Balakrishnan, Vivian Chu, Mauricio A. Hernández, et al. 2010. Midas: Integrating Public Financial Data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, New York, NY, USA, 1187–1190.

[2] Anup Chalamalla, Ihab F. Ilyas, Mourad Ouzzani, and Paolo Papotti. 2014. Descriptive and Prescriptive Data Cleaning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 445–456. https://doi.org/10.1145/2588555.2610520

[3] Adriane Chapman and H. V. Jagadish. 2009. Why Not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 523–534.

[4] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (April 2009), 379–474.

[5] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 2201–2206.

[6] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, and Steven Euijong Whang. 2018. Slice Finder: Automated Data Slicing for Model Validation. arXiv:arXiv:1807.06068

[7] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, et al. 2010. Building Watson: An Overview of the DeepQA Project. *AI Magazine* 31, 3 (Jul. 2010), 59–79. https://doi.org/10.1609/aimag.v31i3.2303

[8] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE Computer Society, USA, 174–185.

[9] Andrew J. Ko, Robin Abraham, Laura Beckwith, et al. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages.

[10] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. ACM, New York, NY, USA, 151–158.

[11] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 15–26.

[12] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-based Bug Localization. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 286–295.

[13] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. The Complexity of Causality and Responsibility for Query Answers and Non-answers. *Proc. VLDB Endow.* 4, 1 (Oct. 2010), 34–45.

[14] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[15] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. ACM, New York, NY, USA, 199–209.

[16] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, Taking Decision Support Benchmarking to the Next Level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. ACM, New York, NY, USA, 582–587.

[17] Sudeepa Roy and Dan Suciu. 2014. A Formal Approach to Finding Explanations for Database Queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 1579–1590.

[18] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1231–1245.

[19] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proc. VLDB Endow.* 6, 8 (June 2013), 553–564.

[20] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-Driven Training Data Debugging for Query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1317–1334. https://doi.org/10.1145/3318464.3389696

[21] Ce Zhang, Christopher Ré, Michael Cafarella, Christopher De Sa, Alex Ratner, Jaeho Shin, Feiran Wang, and Sen Wu. 2017. DeepDive: Declarative Knowledge Base Construction. *Commun. ACM* 60, 5 (April 2017), 93–102.

A APPENDIX

A.1 Explanation Synthesis

At each module for all flows with incorrect outputs, the explanation synthesizer generates all possible atomic predicates that capture some of the incorrect outputs. The synthesizer prunes explanations according to the explanation's support and reach. Intuitively, a good explanation is one with high support. *But what about reach?* An explanation with high reach marks most unknowns as errors. Such an explanation is too pessimistic and, assuming that errors are rare events, an explanation with high reach is less desirable than one with lower reach. An explanation with low reach marks very few unknowns as errors and runs the risk of being too conservative. Therefore, good explanations should have high support and *reasonable* reach.

After generating atomic predicates, the synthesizer then assembles conjunctive pairs or triples, and groups the conjuncts into disjunctive pairs or triples. To keep the explanation set relatively small and to ensure interactive explanation synthesis, the synthesizer prunes the search space and result set as follows:

(1) First, the number of base atoms considered for conjunctions is limited to only those with high support (> MinSupport).
(2) Second, since conjunctions can only decrease support and reach, a conjunction of up to three atoms is only formed if the combined support remains above a minimum support threshold (> MinSupport) and the combined reach decreases, but not below a minimum reach threshold (> MinReach).
(3) Finally, since disjunctions can only increase support and reach, a disjunction of up to three conjuncts is only formed if the combined support increases and reach remains below a maximum reach threshold (< MaxReach).

As an additional optimization, the synthesizer computes support and reach on a module's immediate outputs (using back-propagation to determine the degree of error of an intermediate datapoint) and not the final outputs. This local computation eliminates the overhead of forward-propagating an explanation's labeling of flows to determine its support and reach on the data flow's final outputs.