Arachnid: A Transformation-Oriented Explanation Engine

Sanjay Krishnan [*], Michael J. Franklin [*†], Ken Goldberg [*], Eugene Wu [††]
[*]UC Berkeley,   [†]University of Chicago,   [††]Columbia University
{sanjaykrishnan, franklin, goldberg}@berkeley.edu   ewu@cs.columbia.edu

ABSTRACT

Visual data exploration tools are an important part of exploratory data analysis and enable analysts to easily identify anomalous or surprising patterns and relationships that would otherwise be unnoticed. At this point, the user will want to understand why these anomalies are present, and whether they are due to simple errors or inconsistencies in the dataset, or if they are true patterns. Rather than forcing the analyst to switch into "data cleaning mode", recent work on explanation engines such as Scorpion or MacroBase have proposed automatic algorithms to propose explanations in the form of predicates over the input dataset. If the records matching the predicate were `DELETE`-ed, then the anomalies would disappear. This enables analysts to directly specify anomalies through the visualization and focus their attention on validating candidate explanations. Unfortunately, real-world analysis anomalies require extensions beyond existing explanation engines. Analyses are more complex that SQL aggregation queries, and may instead be machine learning models; real-world datasets exhibit a combinations of different inconsistencies and errors that cannot be resolved by simple `DELETE` transformations; and analysts may want to express complex anomaly patterns such as tuples scored by a machine learning model or string formatting rules.

In this paper, we present Arachnid[1] to address the three above limitations. We model user-specified anomalies as generic quality functions, and explanations as a sequence of parameterized table transformations. We formulate this as a planning problem and propose a generic tree-search algorithm to grow a sequence of table transformations that resolves user-specific anomalies by maximizing the quality function. Although this problem is APX-hard, we show that a combination of pre-specified and adaptively learned pruning rules that exploit structure present in most real-world errors, alongside careful parallelization, allows Arachnid to generate explanations in near-interactive speed.

1. INTRODUCTION

Visual data exploration tools are an important part of exploratory data analysis. These tools provide interfaces to interactively explore subsets, summaries, and processing results of their data through visualizations such as bar charts, scatterplots, line charts, and even summary tables. Visualization is a crucial first-step before formal data modeling and has been widely adopted for understanding machine learning datasets (e.g., Google Facets [1]), data preparation (e.g., Trifacta [4]), business intelligence (e.g., Tableau [42], Spotfire [39]), and have increased interest in the database, visualization, and machine learning communities.

One of the reasons that visualizations are so useful is that analysts can use them to very quickly identify unexpected values, patterns, and trends that would have otherwise been undetected from automated procedures (e.g., the anomaly signatures are subtle or not apriori specified). Consider the following representative examples from two real-world datasets:

EXAMPLE 1 (TERRORISM). *The Global Terrorism Database [18] is a dataset of around terrorist attacks scraped from news sources. Each record contains the date, location, details about the attack, and the number of fatalities and injuries. If we plot the number of incidents for each year, we notice a major increase for 2001. While this is partially explained by the occurrence of 9/11 in the United States, the dataset contains a large number of duplicate records for this year due to increased reporting. There are similarly missing years due to date formatting issues.*

In this example, a histogram can give the analyst broad predicates that might have suspect data, however, to *explain* these anomalies, current users must resort to a combination of manual data inspection and programmatic tools. These unexpected values are not due to real anomalous phenomena but also a combination of inconsistencies in the underlying dataset. This mix between real outliers and data errors requires the analyst to "context switch" into data cleaning mode–to write ad-hoc scripts to handle these issues–and then, switch back the analysis mode to continue. For analysts working with a variety of data sources, it is not scalable to perform manual analysis for anomalies encountered in every dataset.

Recent projects have proposed *explanation engines* that automatically search for candidate explanations for user-defined anomalies. These tools take as input complaint specifications (e.g., whether output attribute values are too high, too low, or otherwise incorrect) and output a ranked list of query predicates most correlated with the anomalies [7,11,37,44]. One way of interpreting these predicates is as exclusion rules–i.e., deleting input records that satisfy the predicates help "resolve" the user-specified complaints with minimal effects on non-complaint results [44].

Going back to the terrorism dataset example, one key limitation of existing explanation engines is that they often only provide a partial explanation for why an aggregate is erroneous if there is dirty data in the mix. Predicate deletion may not be enough to explain output anomalies and may return a trivial or misleading result when there are errors. In Example 1, the dataset contains three classes of errors: duplicates, zero-encoding rather than NULLs, and inconsistencies in location attributes. Rather than predicates that are *correlated* with these anomalies, one would like to know suggested transformations that might make an anomalous aggregate typical–including setting values to defaults, merging entities, etc.

An ideal explanation interface would allow the user to interactively specify the anomalies within the visualization that presented

---

[1]A generalization of Scorpion

(a) Drag NY over NYC to specify duplicates

(b) Drag bar height to specify desired value

(c) Drag city_name over city_code to define FD
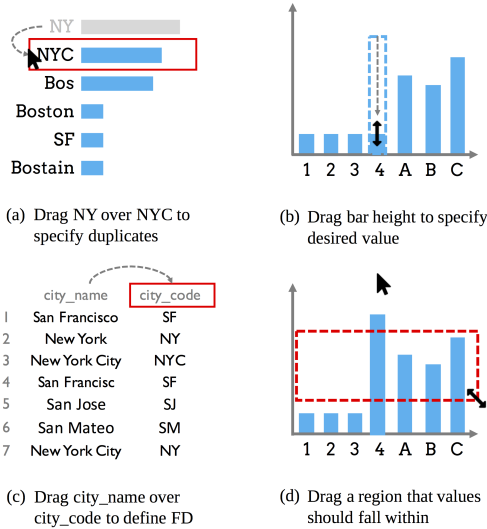
(d) Drag a region that values should fall within

Figure 1: Example interactions to specify visualization anomalies. Arachnid allows a user to specify complaints about a dataset in terms of an arbitrary scoring function and then searches over a language of transformations to best explain those complaints (i.e., what transformations remove them).

the anomalies, and allow an *explanation engine* to generate explanations in the form of succinct sequences of data transformations selected from an extensible library. Furthermore, it should be independent of the visual interface and detection model; effectively allowing for black box complaint specifications.

As a step towards this vision, we present Arachnid, a transformation-oriented explanation engine that generalizes prior explanation engines along three dimensions—analysis program, anomaly types, and explanation language. Arachnid provides a general API to generate explanations in the form of high-level transformation sequences for a wide range of anomalies that can be detected in an output visualization. Transformations are generated from an extensible library of parametrized data transformations. Formally, Arachnid takes as input a *quality function* that scores each output record as a real-value between $[0, 1]$ and a *language* of parameterized data transformation operators, and uses a search-based algorithm to output a sequence of transformations (an explanation) from the language that seeks to maximize the quality function. Similar to prior work [44], the quality functions can be derived from interactions in a visualization interface. For example, one can generate a threshold rule that determines anomalous years in the terrorism dataset. This API imposes minimal restrictions on the quality function, giving it tremendous flexibility regarding the data errors that it can express.

The primary technical challenge is to quickly search the space of possible explanations. This space is combinatorial in the possible transformation sequences as well as the possible parameterizations for each transformation in the sequence. To address this challenge, we make two key observations: 1) most errors are systematic in nature and their structure can be leveraged to reduce the search space, and 2) explanation generation can be cast as a planning problem, where the quality function is the objective and the explanation is the plan, and leverage recent advances in robotics and reinforcement learning [2].

Arachnid uses a best-first search that greedily appends data transformations to a set of best candidate programs seen so far, and adopts parallelization and pruning ideas from the search-based planning literature. In contrast to traditional search problems,

where the search state (e.g., chess board) is compact and largely trivial to parallelize in a distributed setting, the data cleaning search state is the size of the input dataset and introduces a trade-off between communication costs to share intermediate state and the degree of parallelism possible. To further accelerate its runtime, Arachnid can also encode problem-specific optimizations as search pruning rules (e.g., disallowed transformation sequences) or modifications to the data representation (e.g., clustering similar records). Arachnid also leverages the structure of systematic errors to adaptively learn prune rules during the search process, and estimate where or not candidate search branches will ultimately result in high-quality transformation sequences.

The purpose of our experiments is to show the feasibility of a general search-based approach to explanation generation on real-world datasets and analyses. To this end, we evaluate Arachnid on 8 real-world datasets used in prior data explanation and data cleaning literature. In each of these uses cases, we present different classes of quality functions and transformation languages–show that Arachnid is sufficiently general to address all of the domain specific challenges. In addition, we use synthetic datasets and errors to evaluate the precision, recall, and complexity of our proposed explanations in a controlled environment. We quantitatively evaluate Arachnid on its ability to generate accurate transformation rules in terms of accuracy on 7 real case studies. We create best effort pipelines for these case studies with existing systems and compare accuracies and runtimes.

2. RELATED WORK

Arachnid studies the link between data transformation synthesis and explanations. This builds from both data cleaning literature and explanation literature.

Other Explanation Engines: The most common type of explanations are *correlative*. Given a view of the dataset (usually specified in SQL), they identify predicates on a base tables that are most correlated with a selection of unexpected tuples in the view. This model was first proposed by Wu et al. in Scorpion for group-by aggregate visual data analytics, then extended by Roy et al. for more complex SQL queries [37]. Both Wu et al. and Roy et al., consider simple selections of output tuples as the complaint model. Chalamalla et al. studied more complex complaint models where complaints are specified in terms of integrity constraint violations [11]. Finally, Bailis et al. consider this model in the context of streaming anomaly detection [7]. As the age-old axiom goes, correlation does not imply causation, and these systems can have counter-intuitive results when combined with dirty data. We argue that a stronger form of explanations can be generated in many cases where the system can actually suggest transformations (i.e., causation) to rectify anomalies. That said, this new problem is strictly harder and requires much more computation and is a best effort solution sacrificing provable guarantees. This idea is highly related to View Conditioned Causality [30], and we consider what happens when you generalize the search problem to black box analyses.

Data Cleaning Systems: Data cleaning is nearly as old as the relational model [13], and numerous research and commercial systems have been proposed to improve data cleaning efficiency and accuracy (see [34] for a survey). There is certainly an overlap between Arachnid and data cleaning systems. The key difference is Arachnid provides a suggestion of transformation rules rather than directly editing the database instance. The closest neighbor in the data cleaning space is Falcon [22]. Falcon takes human example changes and translates them into transformation rules over the dataset. We explore a higher level of supervision than Fal-

con, where the human simply defines a quality function that scores records as clean or dirty–not actually providing the explicit clean value. Then, the system generates a sequence of transformation rules to best clean the dataset.

However, data cleaning systems provide a good baseline for Arachnid. We validate that the transformations found by Arachnid are sensible by comparing to data cleaning systems designed for similar classes of problems. For example, a quality function can be derived from integrity constraints Arachnid is a much more flexible system that can handle transformations spanning from numerical operations to value replacement. However, Arachnid does not provide a guarantee of constraint satisfaction and therefore cannot be compared in the same class of systems.

Despite this point, we believe that Arachnid is relevant to the recent advances in scalable data cleaning [5,25,27,43]. Recent work has revealed *human-time*—finding and understanding errors, formulating desired characteristics of the data, writing and debugging the cleaning pipeline, and basic software engineering—as a dominant bottleneck in the entire data cleaning process [26]. Arachnid aims to address this bottleneck by using the quality function and transformation language as a flexible and expressive declarative interface to separate high level cleaning goals from how the goals are achieved.

Machine learning has been widely used to improve the efficiency and/or reliability of data cleaning [20,45,46]. It is commonly used to predict an appropriate replacement attribute value for dirty records [45]. Increasingly, it is used in combination with crowd-sourcing to extrapolate patterns from smaller manually-cleaned samples [20,46] and to improve reliability of the automatic repairs [46]. Concepts such as active learning can be leveraged to learn an accurate model with a minimal number of examples [32]. Recently, HoloClean [36] uses probabilistic graphical models to combine multiple quality signals such as lookup tables and constraints to predict cell-level repairs. Although related, Arachnid uses machine learning to steer the search process *away* from low quality candidate programs, rather than to propose ideal cleaning programs. From this perspective, Arachnid can be extended to leverage ideas from existing work to steer the search process *towards* promising programs.

Semantics about the downstream application can inform ways to clean the dataset "just enough" for the application. A large body of literature addresses relational queries over databases with errors by focusing on specific classes of queries [6], leveraging constraints over the input relation [10], integration with crowd-sourcing [9]. Recent work such as ActiveClean [28] extend this work to downstream machine learning applications, while Scorpion [**?**] uses the visualization-specified errors to search for approximate deletion transformations. In this context, Arachnid can embed application-specific cleaning objects can be modeled within the quality function. For instance, our quantitative cleaning experiments (Section 7.4) simply embeds the model training and accuracy computation in the quality function. There has also been recent work on quantifying incompleteness in data quality metrics [12].

Program Synthesis in Data Transformation: A composable transformation language is the building block for systems like Arachnid that generate understandable programs. Languages for data transformations have been well-studied, and include seminal works by Raman and Hellerstein [35] for schema transformations and Galhardas et al. [19] for declarative data cleaning. These ideas were later extended in the Wisteria project [21] to parametrize the transformations to allow for learning and crowdsourcing. Wrangler [24] and Foofah [23] are text extraction and transformation systems that similarly formulate their problems as search over a language of text transformations, and develop specialized pruning rules to reduce the search space. Similarly, BlinkFill uses programming-by-example to generate candidate macros in spreadsheets [41]. In this work, we do not consider the programming-by-demonstrations setting. Our interaction model consists of application interaction which can be turned into a scoring function on base data–again, not giving explicit supervision of what values will optimize the scoring function.

3. PROBLEM DEFINITION

This section presents the basic Arachnid formalisms and our relationship to related work. At a high level, Arachnid is an engine that is given as input **a quality specification** and **a parametrized language of transformations**. Both of these are application specific, but can be designed from common library primitives. Arachnid outputs a sequence of transformations from the language to optimize the quality specification. We find that this program synthesis layer is very valuable for interactive explanations. Since an application, be it a visualization, machine learning model, or SQL query, can output which records seem faulty. For a desired class of updates to the base data, the user can find candidate programs that best rectify these faults.

3.1 Data Transformations

We focus on data transformations that concern a single relational table. Let R be a relation over a set of attributes A, $\mathcal{R}$ denote the set of all possible relations over A, and $r.a$ be the attribute value of $a \in A$ for row $r \in R$. A data transformation $T(R) : \mathcal{R} \mapsto \mathcal{R}$ maps an input relation instance $R \in \mathcal{R}$ to a new (possibly cleaner) instance $R' \in \mathcal{R}$ that is union compatible with R. For instance, "replace all `city` attribute values equal to *San Francisco* with *SF*" may be one data transformation, while "delete records where `city` is *SF*" may be another. Aside from union compatibility, transformations are simply UDFs.

Data transformations can be composed using the binary operator $\circ$ as follows: $(T_i \circ T_j)(R) = T_i(T_j(R))$. The composition of one or more data transformations is informally called an *explanation*, and more formally called a *transformation program p*. If $p = p' \circ T$, then p' is the parent of p; the parent of a single data transformation is a NOOP. In practice, users will specify *transformation templates* $\mathbb{T}(\theta_1, \cdots, \theta_k)$, and every assignment of the parameters represents one possible transformation. Although $\mathbb{T}$ can in theory be an arbitrary deterministic template function, our current implementation makes several simplifying assumptions to bound the number of data transformations that it can output. We assume that a parameter θ_i is typed as an attribute or a value. The former means that θ_i's domain is the set of attribute names in the relation schema; the latter means that θ_i's domain is the set of cell values found in the relation, or otherwise provided by the user.

EXAMPLE 2. *The following* `City` *relation contains three attributes* **city_name**, **city_code**, *and* **population**. *Suppose this data is scraped from the web and contains a number of suspect entries.*

	city_name	city_code	population
1	San Francisco	SF	864,816
2	**New York**	NY	8,538,000
3	New York City	**NYC**	8,491,000
4	**San Francisc**	SF	859k
5	San Jose	SJ	997,811
6	San Mateo	SM	57,194
7	New York City	NY	0

Analyst 1 has a code book `Location`, *that maps a* `city_code` *to a region West, North East, South, Midwest. Suppose the analyst, plotted histograms for each of the regions and their min and max populations. Immediately, she recognizes that some of the values are 0 and not all of the cities are matched with the codebook. The system should be able to search over a set of transformations to rectify this issue (remove as many 0's possible, generate as many code book matches as possible).*

The following transformation template uses three parameters: `attr` *specifies an attribute,* `srcstr` *specifies a source string, and* `targetstr` *specifies a target string.*

$$T = \textit{find\_replace}(srcstr, targetstr, attr)$$

The output of the above is a transformation T that finds all `attr` *values equal to* `srcstr` *and replaces those cells with* `targetstr`. *For instance,* `find_replace("NYC", "NY", "city_code")(City)` *returns a data transformation that finds records in* `City` *whose city_code is "NYC" and replaces their value with "NY". There is additionally a numerical transform that sets the mean over all non-zero entries as default value in the attribute.*

$$T = \textit{set\_default}(srcvalue, attr)$$

There are multiple ways one might define anomalies in this dataset. Analyst 2 might consider using SQL views. She defines a set of functional dependencies creating one-to-one relationships between the city name and code, and an FD relationship between city and population. This defines another type of quality specification. She uses a transformation template that that applies mappings over all possible predicates not just within a single attribute:

$$T = \textit{replace}(src\_predicate, targetstr)$$

We would like to capture both these scenarios in the same general framework

Let Σ be a set of distinct data transformations $\{T_1, \cdots, T_N\}$, and Σ^* be the set of all finite compositions of Σ, i.e., $T_i \circ T_j$. A formal language L over Σ is a subset of Σ^*. A program p is valid if it is an element of L.

EXAMPLE 3. *Continuing Example 2, Σ is defined as all possible parameterizations of* `find_replace`. *Since many possible possible parameterizations are non-sensical (e.g., the source string does not exist in the relation), we may bound Σ to only source and target strings present in each attribute's instance domain (a standard assumption in other work as well [17]). In this case, there are 61 possible data transformations, and Σ^* defines any finite composition of these 61 transformations. The language L can be further restricted to compositions of up to k data transformations.*

Finally, let $Q(R) : \mathcal{R} \mapsto [0, 1]$ be a quality function where 1 implies that the instance R has no anomalies, and a lower value correspond to a more anomalies table. Since running a program $p \in \mathcal{L}$ on the initial dirty table R_{dirty} returns another table, $Q(p(R_{dirty}))$ returns a quality score for each program in the language. Q is a UDF and we do not impose any restrictions on it. In fact, one experiment embeds training and evaluating a machine learning model *within* the quality function (Section 7.4). Another experiment shows that Arachnid can be robust to random noise injected in the function (Section 7.6.2).

Even so, we call out two special cases that provide optimization opportunities. We define two sub-classes of quality functions: row-separable and cell-separable quality functions. The former expresses the overall quality based on row-wise quality function $q(r) : R \mapsto [0, 1]$ where 1 implies that the record is clean: $Q(R) \propto$

$\sum_{r \in R} q(r)$ Similarly, a cell-separable quality function means that there exists a cell-wise quality function $q(r, a) : (R \times A) \mapsto [0, 1]$, such that the quality function is the sum of each cell's quality: $Q(R) \propto \sum_{r \in R} \sum_{a \in A} q(r, a)$.

These special cases are important because they can define hints on what types of transformations are irrelevant. For example, if the quality function is cell-separable, and we have identified that a set of cells C are dirty (e.g., they violate constraints), then we can ignore transformations that do not modify cells in C. This restricts the size the language and makes the problem much easier to solve. We are now ready to present data cleaning as the following optimization problem:

PROBLEM 1 (CLEAN(Q, R_{dirty}, L)). *Given quality function Q, relation R_{dirty}, and language L, find valid program $p^* \in L$ that optimizes Q:*

$$p^* = \max_{p \in L} Q(p(R_{dirty})).$$

$p^*(R_{dirty})$ returns the cleaned table, and p^* can be applied to any table that is union compatible with R_{dirty}. A desirable property of this problem formulation is that it directly trades off runtime with cleaning accuracy and can be stopped at any time (though the cleaning program may be suboptimal). At the limit, Arachnid simply explores L and identifies the optimal program.

EXAMPLE 4. *Continuing Example 2, let us assume the following functional dependencies over the example relation:* `city_name → city_code` *and* `city_code → city_name`. *We can efficiently identify inconsistencies by finding the cities that map to > 1 city code, and vice versa. Let such city names and codes be denoted D_{city_name} and D_{city_code}, respectively. $Q(R)$ is a cell-separable quality function where the cell-wise quality function is defined as $q(r, a) = 1 - (r.a \in D_a)$, such that $r.a$ is 1 if the attribute value does not violate a functional dependency, and 0 otherwise.*

By searching through all possible programs up to length 3 in L, we can find a cleaning program based on `find_replace` *that resolves all inconsistencies:*

```
find_replace(New York, New York City, city_name)
find_replace(San Francisc, San Francisco, city_name)
find_replace(NYC, NY, city_code)
```

3.2 Approach Overview and Challenges

Our problem formulation is a direct instance of *planning* in AI [38], where an agent identifies a sequence of actions to achieve a goal. In our setting, the agent (Arachnid) explores a state space ($\mathcal{R}$) from an initial state (the input relation) by following transitions (applying $T_i \in \Sigma$) such that the sequence of actions is valid (within Σ^*) and the quality of the final state ($Q(R_{final})$) is maximized.

For readers familiar with stochastic processes, this search problem is equivalent to a deterministic Markov Decision Process (MDP), where the states are $\mathcal{R}$, the actions Σ, the transition function updates the instance with the transformation, the initial state is the dirty instance R_{dirty}, and the reward function is Q.

One may be hesitant in adopting our problem formulation because, although it is sufficiently general to model many existing data cleaning problems, such generality often comes at the expense of runtime performance. The planning problem is known to be APX-Hard, meaning there does not exist a polynomial time approximation unless P=NP.

Despite the problem's worst-case complexity, recent successes in similar planning problems—ranging from AlphaGo [40] to automatically playing Atari video games [31] have shown that a prudent combination Machine Learning and distributed search can find
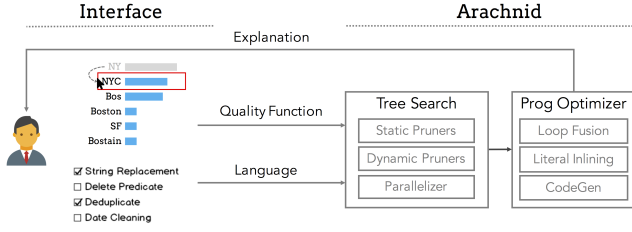
Figure 2: Arachnid architecture. Users interactively specify quality functions through the visualization, and optionally select transformation templates. Arachnid uses an optimized tree search to find and optimize a sequence of transformations (explanation) to maximize the quality measure.

practical solutions by leveraging the structure of the problem. Not every problem instance is as pathological as the worst case complexity suggests, and there are many reasonable local optima.

4. ARCHITECTURE AND API

The *Client Interface* allows the user to interactively specify anomalies through drag-and-drop interactions and lightweight text annotations (Figure **??**). These interactions are translated into a Python quality function Q. Alternatively, the user can write the quality function directly. The interface also provides a list of common transform templates in case the user wants to deselect any that are not applicable to her domain. She may similarly write her own parameterized transformation template. These are used to specify the transformation language L.

The *Tree Search* component takes as input Q and L, and performs a greedy search heuristic to find a program $p^* \in L$ that maximizes Q. To bound the search space, users specify a maximum program length k; Otherwise, Arachnid can continue to run with increasing sizes of k. Arachnid supports three classes optimizations. *Static pruning* invalidates candidate programs based on the program structure (sequence of actions). For instance, composing the same idempotent transformation (e.g., `find_replace(SFO, SF, city_name)`) in succession is unnecessary. *Dynamic pruning* can access the result of a candidate program (search state) when making pruning decisions, and we propose a novel approach to learn automatic dynamic pruning rules that can reduce end-to-end runtimes by up to 75% at the expense of slightly lower recall. Finally, Arachnid *parallelizes* the search in both shared memory and distributed settings. Section 6 describes the pruning rules and parallelization optimization in detail.

Finally, once search component outputs the cleaning program p^*, the *Program Optimizer* performs query compilation optimizations and provides an API to add new optimizations. Arachnid currently uses simple optimization rules: replace variables with literal values whenever possible, inline data transformations into loops that scan over the input relation, and uses loop fusion [14,33] to avoid unnecessary scans over the input and intermediate relations for each transformation in p^*. A text description could be generated for this program, and returned to the user as the candidate explanation.

5. SEARCH OVERVIEW

We now provide an overview of Arachnid's search algorithm and optimizations.

5.1 Naive Search Procedures

In principle, any tree search algorithm over L would be correct. However, the traversal order and expansion policy is important in this search problem. We describe the algorithmic and practical rea-

---

Algorithm 1: Greedy Best-First Tree Search

Data: Q, R, Σ, L, (k, γ)
1 // Initialize priority queue of candidate programs
2 $P = \{NOOP\}$
3 **while** $|\{p \in P \mid p.len < k\}| > 0$ **do**
4   **for** $p \in P : \|p\| < k$ **do**
5   Pop p from the queue.
6   **for** $T \in \Sigma$ **do**
7       $p' = p \circ T$
8   **if** $p' \in L$ **then**
9         $P.push(p', Q(p'(R)))$

10   $\bar{p} = \arg\max_{p \in P} Q(p(R))$
11   $P = \{p \in P \mid p < \gamma \times Q(\bar{p}(R))\}$

12 **return** Highest priority item on the queue

---

sons why two naive procedures—breadth-first search (BFS) and depth-first search (DFS)—exhibit poor search runtimes.

BFS: This approach extends each program in the search frontier with every possible data transformation in Σ. To extend a candidate program l_c with $T \in \Sigma$, it evaluates $Q((T \circ l_c)(R))$. Unfortunately, the frontier grows exponentially with each iteration. Additionally, evaluating every new candidate program $T \circ l_c$ can be expensive if the input relation is large. Although the cost can be reduced by materializing $l_c(R)$, it is not possible to materialize all candidate programs in the frontier for all but the most trivial problems. It is desirable to use asearch procedure that bounds the size of the frontier and the materialization costs.

DFS: Depth-first search only needs to materialize the intermediate results for a single program at a time, however it is highly inefficient since the vast majority of programs that it explores will have low quality scores.

5.2 Search Algorithm and Optimizations

Best-first search expands the most promising nodes chosen according to a specified cost function. We consider a greedy version of this algorithm, which removes nodes on the frontier that are more than γ times worse than the current best solution (Algorithm 1). Making γ smaller makes the algorithm asympotically consistent but uses more memory to store the frontier, whereas $\gamma = 1$ is a pure greedy search with minimal memory requirements.

The frontier is modeled as a priority queue P where the priority is the quality of the candidate program, and is initialized with a NOOP program with quality $Q(R)$. The algorithm iteratively extends all programs in the queue with less than k transformations; a program p is extended by composing it with a transformation T in Σ. If the resulting program p' is in the language L, then we add it to the queue. Finally, let $\bar{p}$ be the highest quality program in the queue. The algorithm removes all programs whose quality is $< \gamma \times Q(\bar{p}(R))$ from the frontier. This process repeats until the candidate programs cannot be improved, or all programs have k transformations.

In a naive and slow implementation, the above algorithm computes p's quality by fully running p on the input relation before running Q on the result, explores all possible data transformation sequences, and runs sequentially. One of the benefits of its simple structure is that it is amenable to a rich set of optimizations to prune the search space, incrementally compute quality functions, and parallelize the search.

Static Pruning Rules are boolean functions that take a candidate

program p as input and decides whether it should be pruned. Arachnid currently models static rules as regular expressions over Σ. Static rules are can be viewed as filters over L.

$$\text{static\_rule}(p) \mapsto \{0, 1\}$$

For example, since the find-and-replace operations are idempotent, i.e., $T(T(R)) = T(R)$, we may want to only consider the set of all sequences with no neighboring repeated transformations. Similarly, we may also want to prune all search branches that make no effect (i.e., find-and-replace New York with New York). These two regular expressions alone reduce the above example's language by 48% (from 226981 to 120050). Other rules, such as avoiding changes that undo previous changes $T^{-1}(T(R)) = R$, are similarly easy to add.

Dynamic Pruning Rules also have access to the input relation and quality function, and can make instance-specific pruning decisions.

$$\text{dyn\_rule}(p, Q, R) \mapsto \{0, 1\}$$

For example, suppose Q is based on functional dependencies and is cell-separable, and we want to ensure that cell-level transformations made by a candidate program p individually improve Q. In this case, we find the cells C that initially violate the functional dependencies and ensure that the cells transformed by p are all in C. Applying this optimization, in addition to the others in Arachnid, to the example reduces the search space by $143\times$ from 226,981 candidate programs to only 1582.

Since it can be challenging to hand-write pruning rules, Section 6.2 describes a dynamic approach that uses simple machine learning models to automatically identify the characteristics of candidate programs to decide whether a particular search brach will be promising. In essence, it generates and refines static pruning rules during the search process.

Divide-and-Conquer: A major cost is that independent errors in the relation must be transformed sequentially in the search algorithm. For instance, records 2, 3, and 4 in Example 2 exhibit independent errors and a fix for a given record does not affect the other records. Thus, if each record were transfored in isolation, the search space would be $O(|\Sigma|)$. Unfortunately, the entire relation requires a program of three transformation to fix the records, which increases the search space to $O(|\Sigma|^3)$.

The main insight in block-wise transformation is that many errors are local to a small number of records. In these cases, it is possible to partition R into a set of blocks $B_1, \cdots, B_m$, execute the search algorithm over each block independently, and concatenate their programs to generate the final transformation program. This gather-scatter approach can exponentially reduce the search space for each block, and reduces the cost of evaluating super-linear quality functions that require e.g., computing a pair-wise similarity scores for the input relation. For example, quality functions derived from functional dependencies can define blocks by examining the violating tuples linked through the dependency. Similarly, users can define custom partitioning functions or learn them via e.g., clustering algorithms. In our current implementation, we partition the input relation by cell or row if the quality function is cell or row separable.

Parallel Program Evaluation: It is clear that the candidate programs can be evaluated and pruning in a parallel fashion across multiple cores and multiple machines, and is one of the major innovations in modern planning systems. However, unlike classic planning problems where communications are small due to the compact size of each state, Arachnid's state size is determined by the relation instance, and can lead to prohibitively high communication costs and memory caching requirements. Section 6.1 describes how we
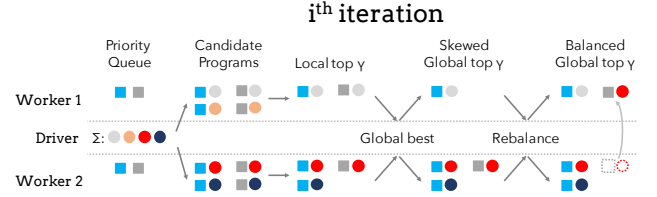

Figure 3: In each iteration, each worker starts with a subset of the priority queue (boxes). The driver sends a subset of data transformations from Σ (circles) to generate candidate programs (box-circles). A series of synchronization points identify the globally top γ candidates and redistributes them across the workers.

manage the trade-offs when parallelizing Arachnid in shared memory and distributed settings.

Materialization: Since each candidate program $p' = p \circ T$ is the composition of a previous candidate program and a transformation, an obvious optimization is to materialize the output of $p(R)$ and incrementally compute $p'(R)$ as $T(p(R))$ over the materialized intermediate relation. Although this works well in a single threaded setting, memory and communication challenges arise when combining materialization with parallelization (Section 6.1).

6.   SEARCH OPTIMIZATIONS

This section describes two important optimizations that allows Arachnid to generate programs efficiently.

6.1   Parallelization

Composing and evaluating $Q(p'(R))$ is the single most expensive operation in the search procedure. We now discuss how we parallelize its evaluation in shared memory and distributed settings, and the challenges when combining it with materialization.

Shared Memory: In a naive, shared-memory implementation, we execute all expansions for a given $p \in P$ in parallel. We materialize $p(R)$ in memory, and evaluate the quality of each $p' = p \circ T \mid T \in \Sigma$ in parallel using a thread pool. Each thread drops a given p' if its quality is lower than $\gamma \times$ the maximum quality from the previous `WHILE` iteration or the local thread. At the end of the `WHILE` iteration, the threads synchronize to compute the highest quality, and flush the remaining candidates using the up-to-date quality value. The output of each $p'(R)$ can be retained or discarded using any cache replacement policy. Our implementation uses Ray [3] to schedule and parallelize the tasks.

Distributed: In the distributed setting, we do not have access to fast shared memory and the communication costs of sharing intermediate relations $p(R)$ can be impractical. Thus, each worker is given a subset of candidate programs to locally evaluate and prune, and the main challenge is to reduce task skew through periodic rebalancing. We use a worker-driver model with j workers (Figure 3).

Let $P^{next} = P \times \Sigma$ be the set of candidate programs (e.g., ■●) to evaluate in the current iteration of the search algorithm. For instance, $P = \{NOOP\}$ in the first iteration, so the candidates are the set of individual data transformations Σ. The driver assigns the input relation R and $\frac{1}{j}$ of P^{next} to each worker. In the figure, the driver assigns a subset of Σ to each worker. Each worker evaluates and computes the top-γ candidates based on the best worker-local quality. The worker runs and caches the parents of its assigned candidate programs (■, ■) to incrementally compute the quality function.

Note that the worker-local top-γ candidates are a superset of the top-γ global candidates because the best local quality is $\leq$ the

global best. Thus the workers synchronize with the driver to identify the global best candidate and further prune each worker's top candidates. At this point, all candidate programs are within γ of the globally best candidate, but their distribution across the workers can be highly skewed. Arachnid performs a final rebalancing step, where each worker sends the number of un-pruned candidates to the driver. Workers with more than $\frac{1}{?}$ of the total number redistribute the extras to workers with too few candidates. When redistributing, workers communicate directly and do not involve the driver (e.g., Worker 2 sends ■● to Worker 1). If the total number is $< k$, then candidates are randomly chosen to be replicated. Only the programs and their qualities are sent; the program results are re-computed by the receiving worker. This ensures that the priority queue in the next iteration is evenly distributed across all workers.

6.2 Dynamically Learning Pruning Rules

To effectively search through the language of transformations, pruning heuristics are important, yet hand-writing such heuristics *a priori* is challenging. We describe how such heuristics can be automatically learned during the search process.

Approach: When Arachnid executes the search algorithm on each block of data, it generates a program that optimizes the quality metric for that block. In many cases, the dataset can be partitioned into a large number of blocks that each serve as sources of training examples for a learned pruning model. Each transformation in a block's program p_b can be labeled as a positive training example in Σ_b^+, while all other transformations serve as negative examples $\Sigma_b^- = \Sigma - \Sigma_b^+$. As Arachnid processes more blocks, the union of these training sets can be sufficient to train a classifier to predict whether a given transformation will be included in the optimal program. In our approach, the prediction model $M(T) : \Sigma \mapsto \{0, 1\}$ is over the data transformations and not the data; is this sense, Arachnid learns static pruning rules in a dynamic fashion.

Internally, Arachnid uses a Logistic Regression classifier that is biased towards false positives (i.e., keeping a bad search branch) over false negatives (e.g., pruning a good branch). This is done by training the model and shifting the prediction threshold until there are no False Negatives.

Featurization: To use this approach, we use featurizers to transform each data transformation T into a k-dimensional feature vector: feat: $T \mapsto \mathbb{R}^k$. For example, recall the `find_replace(NYC, NY, city_code)` transformation in Example 2. The expert is free to encode potential signals as part of the transformation. For instance, the edit distance between the two literal parameters (e.g., NYC, NY) and an indicator vector to specify the attrbitue (e.g., city_code).

Notice that many data transformation can be modeled as a predicate that specifies which records to clean, target attributes to clean, and replacement values for those attributes. Thus, the featurized transformation potentially allows a model to learn which records, which attributes, and what replacement values, are highly to contribute to the final program. For instance, Arachnid may learn that `find_replace` only makes sense for `city_name`. Similarly, it may learn that the replacement string should be similar in edit distance to the source string, and subsequently learn the appropriate edit distance threshould.

Discussion: We believe this is one of the reasons why a simple best-first search strategy can be effective. For the initial blocks, Arachnid searches without a learned pruning rule in order to gather evidence. Over time, the classifier can identify systematic patterns that are unlikely to lead to the final program, and explore the rest of the space. In fact, this incremental learning process can be viewed from an active learning perspective to further target the exploration towards programs that will best improve the classification model. A potential benefit of learning a pruning model for *data transformations* rather than relation instances is that it can potentially be reused or fine-tuned for new, but structurally similar, data transformation problems. In this way, the model can be trained *across data transformation problems* rather than across blocks within a single problem.

7. EXPERIMENTS

7.1 Synthetic Data

In the first set of experiments, we characterize Arachnid on synthetic data. We generate a dataset with the following schema:

```
City(Name, Abreviation, Population)
```

Names are randomly generated from a set of strings scraped from wikipedia, abbreviations are generated from two letters randomly selected from the names, and populations are generated via a Zipfian distribution. This dataset considers multiplicities where there are duplicate (non-erroneous) records.

Anomalies: We consider a number of different anomaly detection criteria:

- **Q1:** All records that violate a one-to-one mapping constraint between `Name` and `Abbrevation`. This is a standard functional dependency constraint to detect anomalies [11].

- **Q2:** All records with a population greater than 6 median absolute deviations from the median population. This is a standard quantitative outlier detection condition to detect anomalies [7,44].

- **Q3:** All records whose abbreviation has a string length not equal to two. This is a more complex UDF that cannot be addressed by existing work.

For each of these anomaly detection criteria, we generate errors in the dataset. For Q1, we select a random subset of each of the records with a particular name and perturb the `Abbrevation` attribute. For Q2, we randomly generate populations with a population of 0 or one that is much higher than typical and replace a subset of records that satisfy a equality predicate on either `Name` and `Abbrevation` or both. For Q3, we randomly add characters to some records abbreviations.

Language: We consider different parametrized languages for transformations:

- **D1:** All deletions with a single attribute equality predicate.

- **D:** All deletions with a multiple attribute equality predicate. This setting of Arachnid is the most similar to prior outlier explanation work.

- **R1:** All single attribute find and replace operations.

- **R:** All multiple attribute find and replace operations. This setting of Arachnid is most similar to prior data cleaning work.

- **M:** All transformations on the population attribute that clip all values above or below a discretized threshold.

7.1.1 Absolute Runtime and Accuracy

Due to the way that we generated errors, we have a natural notion of ground truth. We know the predicates and transformations used the generate errors, and we can calculate the precision and recall. For a generated dataset of 10000 records, we tuned the error generation process such that approximately 5% of records are corrupted. The tables (Table 1 and Table 2) show the F1 score of Arachnid for all combinations of languages and detectors and the runtimes for

	Q1	Q2	Q3	Q123
D1	1.0	0.89	1.0	.88
D	0.84	.86*	1.0	.79
R1	1.0	.71	1.0	.86
R	0.83*	.68	1.0	.77
M	-	1.0	-	-

Table 1: The F1 score of Arachnid for all languages and detector combinations. The starred entries are settings of Arachnid for which a comparable system exists.

	Q1	Q2	Q3	Q123
D1	8.1	3.5	5.4	12.9
D	32.9	24.1*	21.9	59.3
R1	17.5	16.5	17.8	25.7
R	107.8*	59.4	69.3	136.9
M	-	0.17	-	-

Table 2: The runtime of Arachnid for all languages and detector combinations. The starred entries are settings of Arachnid for which a comparable system exists.

each of the pairs in seconds. We use '*' to indicate combinations that are comparable with prior explanation and cleaning systems.

Arachnid provides flexibility to the user the change the language and the quality functions to elict different behaviors. For example, the R and R1 languages are not suited for numerical manipulation. The M language can be used to improve accuracy. The runtime of Arachnid largely depends on how large the support of the language is. More carefully designed languages lead to faster and more accurate transformation programs. More complex quality functions only impact runtime in terms of how expensive they are to compute.

7.1.2 Comparison to Specialized Systems

Next, we compare Arachnid to specialized systems where applicable. First, we compare Arachnid to outlier explanation algorithms when the quality function is Q2 (numerical outliers) and the language is only deletions (D1, D). We consider the following alternatives: Scorpion [44] using a decision tree to extract the predicate, correlation that identifies predicates most correlated with the outliers inspired by and avoids evaluating all subsets of attributes [7], brute force which enumerates all predicates and returns the one with the highest correlation, Arachnid with a single attribute predicate (AC-D), and Arachnid with a multiple attribute predicate (AC). Figure 4 plots the results for F1 score on 10000 records, runtime, and the F1 score as a function of number of records. Arachnid is competitive in terms of accuracy in the specialized cases, but is more expensive in terms of runtime. The run time of Arachnid can be improved by changing the language specification (e.g., restricting the search to a single attribute deletion).

Another intriguing result is the accuracy on a small dataset. Since the alternatives, Scorpion and Brute Force, search over multiple attributes by default. They are susceptible to overfitting to spurious trends when the dataset is small. Arachnid with a single attribute predicate actually achieves an improved accuracy result for a smaller dataset. The flexibility in language can be used as a form of regularization to avoid overfitting.

Next, there is overlap between Arachnid and data cleaning systems. We next compare to standard functional dependency enforcement algorithms on the quality function Q1 (one-to-one relationship) and the replacement languages (R1 and R). We compare Arachnid to a restricted chase algorithm [8] and a tuple-generating dependency-based cleaning system called Llunatic [15]. The challenge is that these systems do not return rules but return cleaned

instances of the database. So, we extract rules from the edits using a decision tree as in scorpion. Then, we compare the systems on two different accuracy metrics: instance accuracy (how well do the transformations recover the true values), and rule accuracy (can we explain the edits with rules).

Figure 5 illustrates the results. While Arachnid is slower than the specialized constraint enforcement systems in terms of runtime, it is competitive in terms of accuracy–especially when it comes to extracting transformation rules. The constraint solving systems operate at a tuple-by-tuple basis and do not necessarily ground their solutions in predicates.

7.2 Real Case Studies

Beyond the characterization, we consider 7 case studies applying Arachnid to real problems. When relevant, we present "best effort" comparisons with existing systems–as in the synthetic experiments.

Flight: The flight dataset [16] contains arrival time, departure time, and gate information aggregated from 3 airline websites (AA, UA, Continental), 8 airport websites (e.g., SFO, DEN), and 27 third-party websites. There are 1200 flight departures and arrivals at airline hubs recorded from each source. Each flight has a unique and globally consistent ID, and the task is to reconcile data from different sources using the functional dependency ID→ arrival, departure, gate information.

FEC: The election contributions dataset has 6,410,678 records, and 18 numerical, discrete, and text attributes. This dataset records the contribution amount and contributor demographic information e.g., name, address, and occupation. The task is to enforce city→zipcode, and match occupation to a codebook on canonical occupations. The quality function is 1 if the occupation is in the codebook, and 0 otherwise; we penalize the edit distance between the original and edited occupation values.

Malasakit: This dataset contains 1493 survey disaster preparedness responses from the Philippines, with 15 numeric and discrete attributes. The task removes improper numerical values and remove dummy test records. This consists of domain integrity constraints that force the values to be within a certain dictionary.

Physician: This dataset from Medicare.gov contains 37k US physician records, 10 attributes, with spelling errors in city names, zipcodes, and other text attributes. We use the rules described in [36], which consists of 9 functional dependencies for error detection.

Census: This US adult census data contains 32k records, and 15 numeric and discrete attributes. There are many missing values coded as 999999. The task is to clean numeric values to build a classification model that predicts whether an adult earns more than $50k$ annually.

EEG: The 2406 records are each a variable-length time-series of EEG readings (16 numeric attributes), and labeled as "Preictal" for pre-seizure and "Interictal" for non-seizure. The goal is to predict seizures based on 32 features computed as the mean and avriance of the numeric EEG attributes. The task is to identify and remove outlier reading values.

Terrorism: The Global Terrorism Database [18] is a dataset of around terrorist attacks scraped from news sources. Each record contains the date, location, details about the attack, and the number of fatalities and injuries. The dataset contains a mixture of data errors: (1) there are many duplicates for the same terrorist incident, (2) many missing values in the fatalities and injuries attributes are encoded as zeros, which overlaps with attacks that did not have any fatalities/injuries, and (3) the location attributes are inconsistently encoded. We used the dataset from 1970, and there are
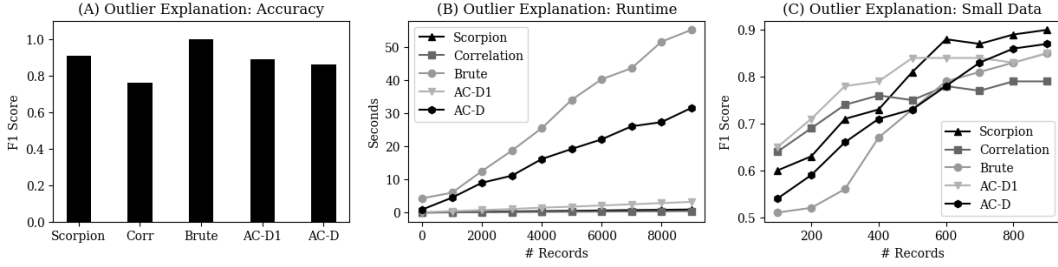
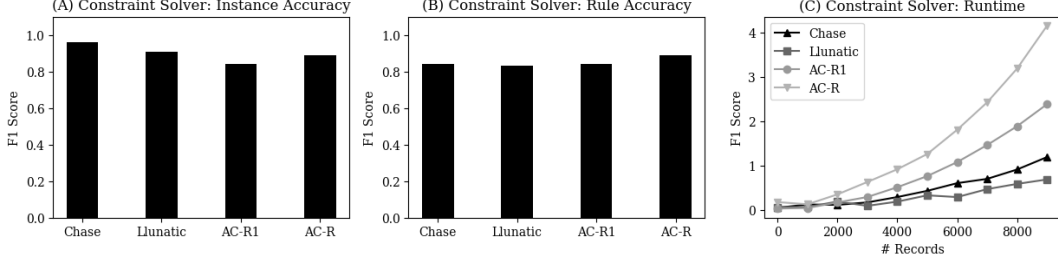Figure 4: Comparison with outlier explanation systems.
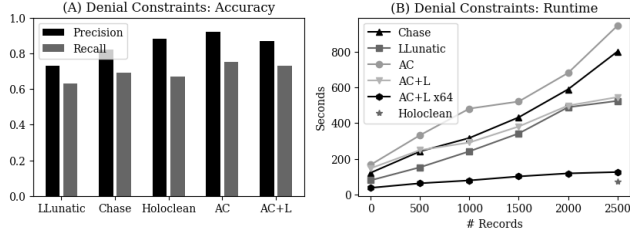

Figure 5: Comparison with FD repair systems.

Figure 6: Comparison with denial constraint systems on the Flight dataset. (A) Arachnid (AC) matches or exceeds the accuracy of the specialized systems. (B) The structure of the data errors, along the learning and parallelization (AC+L x64) lets Arachnid scale sub-linearly and outperform all but HoloClean's reported runtimes.

170000 records. We downloaded this dataset and sought to understand whether terrorist attacks have become more lethal than they were in the 1970s. To do so, we hand cleaned the records to create a gold standard. It turns out that, in this dataset, attacks have become more lethal, but fewer in number than 50 years ago. This task was intentionally open-ended to represent the nature of the iterative analysis process.

7.3 Constraint Violations

Denial constraints express a wide range of integrity constraints and form the basis of many data cleaning systems. We can compose a quality function that quantifies the number of constraint violations and find transformations that reduce the number of violations. For Arachnid, we search over single attribute find and replace transformations (R1 in the synthetic experiments). We quantify the precision and recall of the transformations discovered. We use the Flight dataset for these experiments.

Baselines: We run against (1) Llunatic, a denial constraint-based cleaning system [15] implemented in C++ on top of PostgreSQL[2], and (2) a restricted chase algorithm [8] implemented in Python. We compare against the chase because a large portion of denial constraints are functional dependencies, and can be resolved using fixed-point iteration. We report numbers from the recent Holoclean publication [36] that used the same datasets and constraints, but did not run the experiment ourselves.

_____
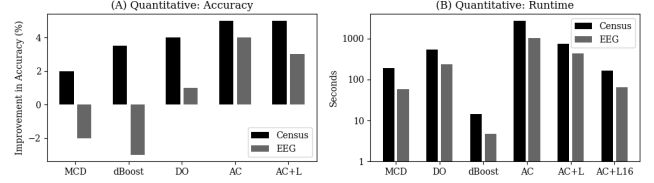[2]Constraints are specified as Tuple-Generating Dependencies

Figure 7: Numerical Transformations on census and EEG datasets for a classification application. Arachnid tranformation can clip outliers or set them to a default value. (A) Arachnid has a higher accuracy than outlier detection algorithms (MCD, dBoost), and Arachnid with a single transform template (DO). (B) Optimizations improve Arachnid runtime by over an order of magnitude.

Results: Figure 6a shows the precision and recall of each approaches based on known ground truth. Arachnid matches or beats the accuracy of the baselines, however its runtime (AC) without any learning scales poorly compared to alternatives (Figure 6b). Using learning (AC+L) shows performance on par with LLunatic, and parallelization on 64 threads is comparable to Holoclean's reported runtime. The results suggest that learning exhibits sublinear scaling due to Arachnid learning more effective pruning rules as it sees more data. These performance gains are at the expense of slightly reduced accuracy.

We also evaluated Arachnid (single threaded, without learning) on the FEC, Malasakit, and Physician datasets. Their precision, recall, and runtimes are as follows: FEC: 94% prec, 68% rec, 5hrs; Malasakit: 100% prec, 85% rec, 0.39hrs; Physician: 100% prec, 84%, 3.4hrs.

7.4 Numerical Transformations

This experiment performs numerical transformations on machine learning data. We explore to what extent the transformations provided by Arachnid can be used to improve the performance of a downstream data product such as machine learning. In these applications, prediction labels and test records are typically clean and available (e.g., results of a sales lead), whereas the training features are often integrated from disparate sources and exhibit considerable noise (e.g., outliers). Our quality function is simply defined as the model's accuracy on a training hold-out set, and we report the test accuracy on a separate test set.

We trained a binary classification random forest model using

`sklearn` on the Census and EEG datasets. We used standard featurizers (hot-one encoding for categorical data, bag-of-words for string data, numerical data as is) similar to [20]. We split the dataset into 20% test and 80% training, and further split training into 20/80 into hold-out and training. We run the search over the training data, and evaluate the quality function using the hold-out. Final numbers are reported using the test data.

We defined the following three data transformation templates that sets numerical attribute values in R if they satsify a predicate:

- **clip_gt(attr, thresh):** $R.attr = thresh$ if $R.attr > thresh$
- **clip_lt(attr, thresh):** $R.attr = thresh$ if $R.attr < thresh$
- **default(attr, badval):** $R.attr$ set to mean val if $R.attr = badval$

Baselines: We compare with 4 baselines: *No Transformation (NC)*, *Minimum Covariance Determinant (MCD)* is a robust outlier detection algorithm used in [7] and sets all detected outliers to the mean value, *dBoost* uses a fast partitioned histogram technique to detect outliers [29], and *Default Only (DO)* runs Arachnid with only the default() transformation.

Results: The classifer achieves 82% and 79% accuracy on the uncleaned census and EEG data, respectively. Most outliers in the census data are far from the mean, so MCD and dBoost can effectively find. Further, setting census outliers to the mean is sensible. However, the same fix is not appropriate for the EEG data; it is better to clip the outlier values, thus MCD, dBoost, and DO have negligible or negative impact on accuracy. When we realized this from running DO, it was straightforward to add the clipping transformations to the language, and with no other changes, re-run Arachnid with drastically improved EEG accuracies.

Vanilla Arachnid (AC) is nearly $10\times$ slower than MCD, but the adding learning and 16-thread parallelization matches MCD's runtimes. dBoost is specialized for fast outlier detection and Arachnid is unlikely to match its runtime.
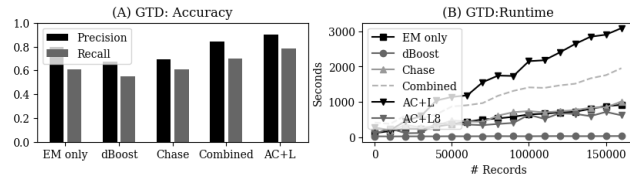

Figure 8: The Global Terrorism Database is a dataset of terrorist attacks scraped from news sources since 1970. (A) Shows that Arachnid can integrate many different forms of transformations that were previously handled by disparate systems, (B) Arachnid achieves a competitive runtime to using all of the different system and accounting for data transfer time between them.

7.5 Mixing Error Classes

This experiment, we use the multi-error Terrorism dataset [18].

Baselines: We hand-coded blocking-based entity matching (EM) and restricted chase (FD) algorithms in Python, and used dBoost for missing values. We also ran EM, dBoost, and Chase serially on the dataset (Combined). We use that combination to generate transformation explanations.

Results: Figure 8a shows that combining all three classes of errors within the same Arachnid framework (AC) achieves higher precision and recall than all baselines. In fact, the combined baselines still does not achieve Arachnid's level of accuracy because the operations need to be interleaved differently for different blocks. Although Arachnid is slower than any individual baseline, parallelizing Arachnid to 8 threads is faster than the combined baseline by 2x.

7.6 Arachnid In Depth

This subsection uses the FEC setup to study the parameters that affect Arachnid's accuracy and runtime, the robustness of its transformation programs, and its algorithmic properties.

7.6.1 Algorithmic Sensitivity

Partitioning: Partitioning the dataset into smaller blocks effectively reduces the problem complexity. This can have tremendous performance benefits when each block exhibits very few errors that are independent of the other blocks. **??**a shows the performance benefits when varying the block size; we define the blocks by partitioning on three different attributes that have different domain sizes. Reducing the block size directly improves the runtime; the search is effectively non-terminating when blocking is not used.

Language: **??**b fixes the input to a single block, and evaluates the runtime based on the size of the language $|\Sigma|$. Increasing the transformations increases the branching factor of the search problem. The search time is exponential in the language, however Arachnid's learning optimization can identify a pruning model that reduces the runtime to linear.

Coupling in the Quality Function: The complexity of the quality function directly affects search time. A cell-separable quality function is the simplest to optimize because each cell in the relation can be analyzed and cleaned in isolation. In contrast, a quality function that couples multiple records together is more challenging to optimize because a longer sequence of transformation may be needed to sufficiently clean the records and improve the quality function.

We evaluate this by artificially coupling between 1-10 records together, and creating a quality function that only improves when an attribute of the coupled records all have the same value. We perform this coupling in two ways: *Random* couples randomly selected records, whereas *Correlated* first sorts the relation an attribute and couple records within a continuous range. We expect that the random coupling requires individual operations for each record based on their IDs, whereas the correlated setting both allows Arachnid to exploit the correlated structure to learn effective pruning rules and to clean the coupled records using a single operation. **??**c shows that this is indeed the case when running Arachnid on a single fixed-size block. *Random* slows down exponentially with increased coupling, whereas *Correlated* increases linearly.

Quality Function Complexity: Finally, we incrementally increase the quality function's complexity and show haw it affects the accuracy. We add the following constraints in sequence: one functional dependency (FD), a second FD, an entity resolution similarity rule, and a third FD. We define the quality function as the sum of each constraint's quality function. **??**d shows that the F1 accuracy decreases as more constraints are added, however the F1 score is still above 75%.

7.6.2 Generalization and Overfitting

An important characteristic of generating transformation solutions as *programs* is that we can evaluate the program's robustness in terms of machine learning concepts such as overfitting and generalization. To this end, we examine two concepts in the context of data transformations: regularizaton and overfitting. We also find that Arachnid's high level interface is helpful for iteratively tuning the transformation process.

Regularization: Misspecified quality functions can cause Arachnid to output poorly performing transformation programs. We simulate this by adding random noise to the output of the quality function. Figure 9A plots the F1-score of Arachnid on the FEC experiment while varying the amount of noise. As expected, the output
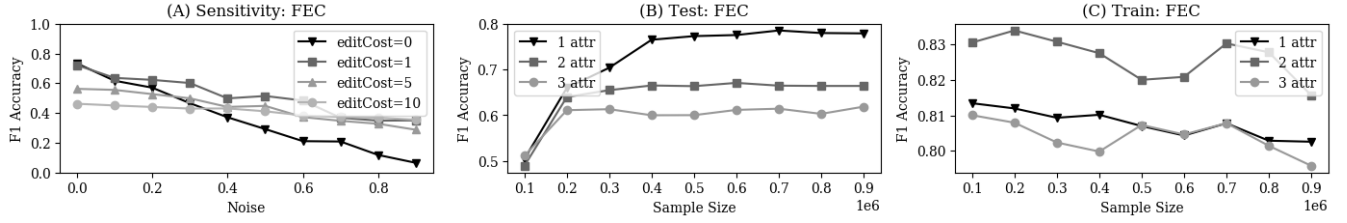
Figure 9: (A) Regularization by increasing an editCost penalty makes Arachnid more robust to noisy (mis-specified) quality functions, (B-C) overly expressive transformation templates can lead to overfitting (2 attr) or an infeasibly large search space (3 attr).
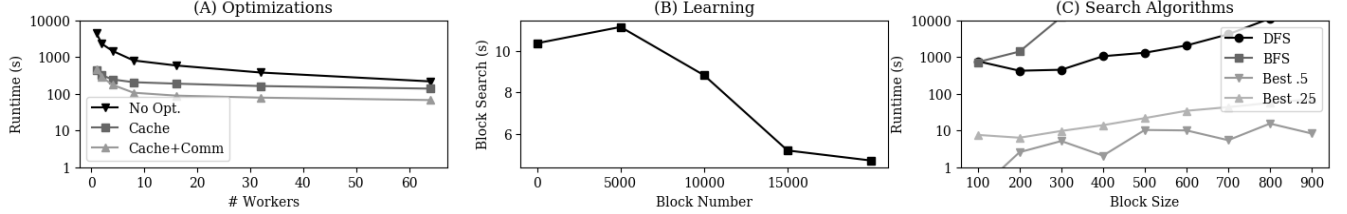

Figure 10: (A) Both the materialization (Cache) and distributed communication (Comm) optimizations contribute to improved scale-out runtimes. (B) The learned pruning rules improve the search costs for each subsequent block-wise partition. (C) Best-first search is better than BFS and DFS; reducing γ prunes more candidates at the expense of lower accuracy.

program's F1-score rapidly degrades as the noise increases.

Machine learning uses regularizing penalty terms to prevent overfitting. We can similarly add a penalty to the quality function to prevent too many edits. Each line line shows the edit cost penalty and shows that although the F1 is lower when there is no noise, Arachnid is more robust to larger amounts of noise.

Overfitting: In machine learning, over-parameterized models may be susceptible to overfitting. A similar property is possible if the language Σ is overly expressive. We use a transformation template that finds records matching a parameterized predicate and sets an attribute to another value in its instance domain. We then vary the language expressiveness by increasing number of attributes in the predicate between 1 and 3. Finally, we run Arachnid on a training sample of the dataset (x-axis), and report F1 accuracy on the training and a separate test sample (Figure 9B-C). Note that overfitting occurs when the training accuracy is significantly higher than test accuracy.

Indeed we find an interesting trade-off. The 1 attribute predicate performed worst on the training sample but outperformed the alternatives on the test sample. The 2 attribute predicate was more expressive and overfit to the training data. Finally, the 3 attribute predicate is overly expressive and computationally difficult to search. Thus, it did not sufficiently explore the search space to reliably identify high quality transformation programs.

Discussion: We have shown that data transformations can overfit, and believe this is a potential issue in *any* transformation procedure. These results highlight the importance of domain experts to judge and constrain the problem in ways that will likely generalize to future data. Further, it shows the value of a high-level interface that experts can use to express these contraints by iteratively tuning the quality function and transformation language.

7.6.3 Scaling

Next, we present preliminary results illustrating the scaling properties of Arachnid.

Parallelization Optimizations: The experiments run on a cluster of 4 mx.large EC2 instances, and we treat each worker in a distributed (not shared-memory) fashion. Figure 10a shows the benefits of the materialization and communication optimizations in Section 6.1. *No opt.* simply runs best-first search in parallel without

any materialization; workers only synchronize at the end of an iteration by sending their top-γ candidate programs to the driver, which prunes and redistributes the candidates in the next iteration. *Cache* extends No opt by locally materializing parent programs, and *Cache+Comm* further adds the communication optimizations for distributed parallelization.

The single threaded No Opt setting runs in $4432s$, and the materialization optimization reduces the runtime by $10\times$ to $432s$. Scaling out improves all methods: at 64 workers, *Cache+Comm* takes 67s while *Cache* takes 137s. Surprisingly, although *No Opt* with 64 workers is slower than *Cache+Comm* by $10\times$, it scales the best because it only synchronizes at the end of an iteration and only communicates candidate programs and their quality values. In contrast, the alternative methods may communicate materialized relation instances.

Within-Block Learning: Although we have shown how learning reduce the overall search runtime, we show that learning also improves the search speed for individual blocks. We run single-threaded Arachnid and report the time to evaluate each block. Figure 10b shows the ith block that is processed on the x-axis, and the time to process it on the y-axis. We see that as more blocks are cleaned, the learned pruning classifier is more effective at pruning implausible candidate programs. This reduces the per-block search time by up to 75%.

Search Algorithm Choice: Figure 10c shows that best-first search out-performs naive depth and breadth first search. We also report Arachnid when $\gamma = \{0.5, 0.25\}$. We see that as the block size increases, DFS and BFS quickly become infeasible, whereas Arachnid runs orders of magnitude more quickly. In addition, reducing γ improves the runtime, however can come at the cost of reduced accuracy by pruning locally sub-optimal but globally optimal candidate programs.

7.6.4 Program Structure

Finally, we present results describing the structure of the data transformation programs found with Arachnid. It is often the case that the program found by Arachnid is a concise description of the needed data transformation operations, that is, the total number of cell edits is much larger than the length of the program. We consider the FEC dataset, the EEG dataset, and GTD dataset.

Sometimes, the program (FEC and GTD) encodes a significant

amount of literal values. This happens in entity matching type problems. For these problems, the program length is relatively large, however, the number of cells modified is even larger (up to 10x more). For datasets like the EEG dataset, the program is very concise (26x smaller than the number of cells modified). Numerical thresholds are generalize better than categorical find-and-replace operations.

	Program Length	Cells Modified
FEC	6416	78342
EEG	6	161
GTD	1014	104992

8. CONCLUSION AND FUTURE WORK

In summary, Arachnid poses explanation generation as a planning problem over a language of data transformations, and presents existing error-specific approaches—constraint-based, statistical, and demonstration-based data cleaning—within a single framework. Our results suggest that borrowing from recent advances in planning and optimization is a fruitful direction. While we focus on explanation in this work, we are excited to extend Arachnid towards data cleaning system. In particular, data cleaning is inherently a visual and interactive process, and we plan to integrate Arachnid with a data visualization interface. Users can visually manipulate and examine their dataset and the system can translate interactive manipulations into quality functions. This will also require work to characterize failure modes and provide high level tools to debug such cases. We are also hopeful that the compact codebase (<200 LOC for the core search and learning algorithms) can enable more rapid development of specialized data cleaning systems for novel domains and error conditions.

9. REFERENCES

[1] Facets - know your data. https://pair-code.github.io/facets/.
[2] Google deepmind. https://deepmind.com.
[3] Ray: A high-performance distributed execution engine. https://github.com/ray-project/ray.
[4] Trifacta. https://www.trifacta.com/.
[5] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. In *VLDB*, 2014.
[6] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: a framework for integrating entity resolution with query processing. In *VLDB*. VLDB Endowment, 2015.
[7] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556. ACM, 2017.
[8] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, 2017.
[9] M. Bergman, T. Milo, S. Novgorodov, and W. C. Tan. Query-oriented data cleaning with oracles. In *SIGMOD*, 2015.
[10] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.
[11] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, 2014.
[12] Y. Chung, S. Krishnan, and T. Kraska. A data quality metric (dqm): How to estimate the number of undetected errors in data sets. 2014.
[13] E. F. Codd. A relational model of data for large shared data banks. In *Communications of the ACM*. ACM, 1970.
[14] A. Crotty, A. Galakatos, and T. Kraska. Tupleware: Distributed machine learning on small clusters. In *IEEE Data Eng. Bull.*, 2014.
[15] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, 2013.
[16] X. L. Dong. Data sets for data fusion experiments. http://lunadong.com/fusionDataSets.htm.
[17] W. Fan and F. Geerts. *Foundations of Data Quality Management*. 2012.
[18] N. C. for the Study of Terrorism and R. to Terrorism (START). Global terrorism database [data file]. https://www.start.umd.edu/gtd/, 2016.
[19] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *PVLDB*, 2001.
[20] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.
[21] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. In *VLDB*, 2015.
[22] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning: A tossed stone raises a thousand ripples. In *Proceedings of the 2016 International Conference on Management of Data*, pages 893–907. ACM, 2016.
[23] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.
[24] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.
[25] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, 2015.
[26] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu. Towards reliable interactive data cleaning: A user survey and recommendations. In *HILDA*, 2016.
[27] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu. Sampleclean: Fast and reliable analytics on dirty data. In *IEEE Data Eng. Bull.*, 2015.
[28] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. In *PVLDB*, 2016.
[29] Z. Mariet, R. Harding, S. Madden, et al. Outlier detection in heterogeneous datasets using automatic tuple expansion. 2016.
[30] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 505–516. ACM, 2011.
[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. In *Nature*. Nature Research, 2015.
[32] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. In *VLDB*, 2014.
[33] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.
[34] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. In *IEEE Data Eng. Bull.*, 2000.
[35] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.
[36] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. In *arXiv*, 2017.
[37] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment*, 9(4):348–359, 2015.
[38] S. Russell, P. Norvig, and A. Intelligence. A modern approach. In *AI, Prentice-Hall*, 1995.
[39] B. Shneiderman. Dynamic queries, starfield displays, and the path to spotfire. 1999.
[40] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam,

M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. In *Nature*. Nature Research, 2016.

[41] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

[42] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *TVCG*, 2002.

[43] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, 2014.

[44] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *VLDB*, 2013.

[45] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[46] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. In *VLDB*, 2011.