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

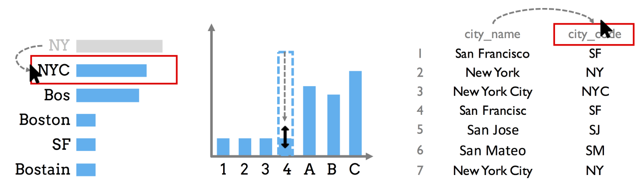
Existing data cleaning systems are specialized to particular classes of data errors such as statistical outliers, constraint violations, and duplicate entities. Yet, any individual data set can exhibit can have many different types of data errors. Managing a pipeline composed of many disparate systems is sufficiently difficult to maintain, debug, and interpret, that many data users opt to write custom scripts instead. Rather than focus on a specific class of errors, we propose to view all data cleaning problems within a single unified abstraction: sequential search over a language of allowable data transformations to maximize an objective function that encodes a notion of "cleanliness". This general problem is often intractable in the worst case, but recent AI successes such as Google's AlphaGo have shown that a combination of machine learning and parallelized search can solve planning problems previously considered impractical. We present Arachnid, a general data cleaning systems which borrows these insights to support arbitrary combinations of existing and new data errors. We evaluated Arachnid against special purpose systems over 8 different datasets. When there is a single error type, Arachnid matches or exceeds the accuracy of special purpose approaches and is camparable in runtime. On multi-error datasets, Arachnid has 10% higher accuracy than a naive combination of specialized systems and runs over 2× faster. Finally, our optimizations can reduce the basic search algorithm from exponential to linear runtime.

1. INTRODUCTION

Visual data exploration tools simplify the process for analysts and data scientists to interactively explore subsets, summaries, and processing results of their data through visualizations such as bar charts, scatterplots, line charts, and even summary tables. These tools have been widely adopted for understanding machine learning datasets (e.g., Google Facets [1]), data preparation (e.g., Trifacta [3]), business intelligence (e.g., Tableau [36], Spotfire [34]), and have increased interest in the database, visualization, and machine learning communities.

Visualizations are particularly powerful for 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 [16] 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 analyst analyzes this dataset to understand whether terrorist attacks have become more lethan than they were in the 1970s.*



(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

Figure 1: Example interactions to specify visualization anomalies.

{{ewu:} Describe the anomaly!}

EXAMPLE 2 (INTEGRATION). *DESCRIBE THE ML OR INTEGRATION EXAMPLE*

In both examples, it is easy for an analyst to detect anomalous patterns in the visualized results, however in order to *explain* these anomalies, current users must resort to a combination of manual data inspection and programmatic tools. Although analysis may be capable of performing detailed manual analysis, the context switch interrupts their the visualization exploration session. Further, analysts are often working with a variety of data sources, and it is not scalable to perform manual analysis for anomalies encountered in every dataset.

To make debugging such errors more efficient, recent projects have proposed *explanation engines* that automatically search for candidate explanations for user-psecific anomalies. Specifically, 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 [6,10,32,38]. This means that deleting inptu records that satisfy the predicates help "resolve" the user-specified complaints with minimal effects on non-complaint results [38]. These systems enable analysts to iterate more efficiently and determine whether sets of unexpected values are due to dirty data or are truly anomalous real phenomena.

The key limitation of existing explanation engines is that they are designed under assumptions of a narrow class of analysis programs, output anomalies, and explanations. More specifically, current engines support analysis programs that are SQL aggregation queries, anomalies as value errors the output records, and explanations in the form of a single predicate deletion transformations.

However in practice, all three assumptions may not hold. First, analyses can consist of SQL queries, machine learning models, or **{{ewu:} XXX}**. Second, output anomalies may also include duplicate identification, identifying the presence (or lack) of correlations between attributes in the results, low training accuracy, or more. Finally, datasets can contain a wide range of systematic errors. Predicate deletion may not be enough to explain output anomalies, and a combination of data transformations (e.g., string formatting, outliers removal, missing values imputation, deduplication) may be needed to fully explain the anomaly. In Example 1, the dataset contained three classes of errors: terrorist incidents are often duplicated multiple times, many missing values in the fatalities and injuries attributes are erroneously encoded as zeros rather than NULLs, and location attributes are inconsistently encoded. Transformations to address all three issues were needed in order to analyze the results. Similarly, the data in Example 2 **{{ewu:} EXHIBITS STUFF}**.

An ideal explanation interface would allow the user to interactively specify the anomalies within the visualization that presented the anomalies, and allow an *explanation engine* to generate explanations in the form of succinct sequences of data transformations selected from an exensible library. **{{ewu:} Give an example sequence for terrorists}**. In many settings, the analyst may use their domain expertise to limit the classes of data transformations to those that make sense for her use case. **{{ewu:} For example, XXX}**

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. It 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, and supports an extensible library of data transformations. Specifically, 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 [38], the quality functions can be derived from interactions in a visualization interface. **{{ewu:} Example?}** This API imposes minimal restrictions on the quality function, giving it tremendous flexibility in terms of 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 [].

To this end, 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 langauages–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. **{{ewu:} We quantitatively evaluate Arachnid on its ability to generate accurate transformation rules in terms of precision and recall.}**

2. RELATED WORK

{{ewu:} what is different from data cleaning? and why you shouldn't use a cleaning system for explanations and call it a da}

Data cleaning is nearly as old as the relational model [12], and numerous research and commercial systems have been proposed to improve data cleaning efficiency and accuracy (see [29] for a survey and Section 3.2 for related systems). The recent advances in scalable data cleaning [4,22,24,37] 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 [23]. 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 in Data Cleaning: Machine learning has been widely used to improve the efficiency and/or reliability of data cleaning [18,39,40]. It is commonly used to predict an appropriate replacement attribute value for dirty records [39]. Increasingly, it is used in combination with crowd-sourcing to extrapolate patterns from smaller manually-cleaned samples [18,40] and to improve reliability of the automatic repairs [40]. Concepts such as active learning can be leveraged to learn an accurate model with a minimal number of examples [28]. Recently, HoloClean [31] 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.

Application-Aware Cleaning: 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 [5], leveraging constraints over the input relation [9], integration with crowd-sourcing [8]. Recent work such as ActiveClean [25] 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 7.2.2) 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 [11].

Generating Cleaning Programs: A composable data cleaning language is the building block for systems like Arachnid that generate understandable cleaning programs. Languages for data transformations have been well-studied, and include seminal works by Raman and Hellerstein [30] for schema transformations and Galhardas et al. [17] for declarative data cleaning. These ideas were later extended in the Wisteria project [19] to parametrize the transformations to allow for learning and crowdsourcing. Wrangler [21] and Foofah [20] 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. These can be viewed as special cases of Arachnid.

3. PROBLEM DEFINITION

First, we overview the basic formalism of Arachnid and present its relationship to related work.

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 the 10^{th} record" 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 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 3. *The following `City` relation contains two attributes city_name and city_code. Suppose there is a one-to-one relationship between the two attributes. In this case, the relation is inconsistent with respect to the relationship and contains errors highlighted in red.*

	city_name	city_code
1	San Francisco	SF
2	**New York**	NY
3	New York City	**NYC**
4	**San Francisc**	SF
5	San Jose	SJ
6	San Mateo	SM
7	New York City	NY

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".

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 4. *Continuing Example 3, 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 [15]). 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 7.2.2). Another experiment shows that Arachnid can be robust to random noise injected in the function (Section 7.4.2).

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 5. *Continuing Example 3, let us assume the following functional dependencies over the example relation: city_name $\rightarrow$ city_code and city_code $\rightarrow$ 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 Problem Expressiveness

3.3 Case Studies

TODO

3.4 Approach Overview and Challenges

Our problem formulation is a direct instance of *planning* in AI [33], 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 APX-Hard, meaning there does not exist a polynomial time approximation unless P=NP. Let R be a single-attribute relation of Booleans. Let L be the set of all assignments to a single value. Given a list of N Boolean clauses over all the boolean variables, let Q assign to each record one minus the fraction of clauses that evaluate to true. This formulation is equivalent to MAX-SAT and solution to the optimization problem.

Despite the problem's worst-case complexity, recent successes in similar planning problems—ranging from AlphaGo [35] to automatically playing Atari video games [27] have shown that a prudent combination Machine Learning and distributed search can find 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

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

Algorithm 1: Greedy Best-First Tree Search

Data: Q, R, Σ, L, (k, γ)

// Initialize priority queue of candidate programs
$P = \{NOOP\}$
while $|\{p \in P \mid p.len < k\}| > 0$ **do**
  for $p \in P : \|p\| < k$ **do**
    Pop p from the queue.
    for $T \in \Sigma$ **do**
      $p' = p \circ T$
      if $p' \in L$ **then**
        $P.push(p', Q(p'(R)))$
  $\bar{p} = \arg\max_{p \in P} Q(p(R))$
  $P = \{p \in P \mid p < \gamma \times Q(\bar{p}(R))\}$
return Highest priority item on the queue

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× 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 3 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 manage the trade-offs when parallelizing Arachnid in shared memory and distributed settings.

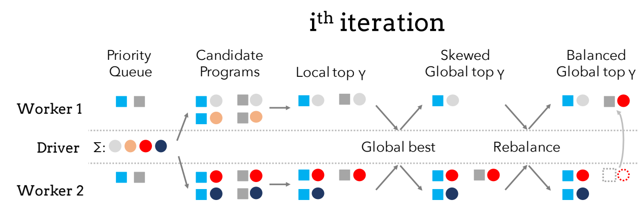


Figure 2: In each iteration, each worker starts with a subset of the priority queue (boxes). The driver sends a subset of data transformations from Σ (circles) to generate candidate programs (box-circles). A series of synchronization points identify the globally top γ candidates and redistributes them across the workers.

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 [2] 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 2).

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 $\frac{1}{j}$ 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: $\textsf{feat}: T \mapsto \mathbb{R}^k$. For example, recall the `find_replace(NYC, NY, city_code)` transformation in Example 3. 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

Next, we present experimental results that suggest three main conclusions: (1) as a single framework, Arachnid can achieve parity in terms of accuracy with state-of-the-art approaches to a variety of different problems ranging from integrity constraint satisfaction, statistical data cleaning, and also data cleaning for machine learning, (2) it is possible to significantly reduce the runtime gap between Arachnid and specialized frameworks using simple pruning and distributed parallelization techniques, and (3) Arachnid enables automatic data cleaning for datasets containing a mixture of error types in an acceptable amount of time.

7.1 Datasets and Cleaning Tasks

We list the main characteristics of the 8 experimental datasets.

Flight: The flight dataset [14] 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`$\rightarrow$ `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`$\rightarrow$`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 physican records, 10 attributes, with spelling errors in city names, zipcodes, and other text attributes. We use the data cleaning rules described in [31], which consists of 9 functional dependencies.

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.

Stock: There are 1000 ticker symbols from 55 sources for every trading day in a month [14]. The cleaning task involves (1) integrating the schemas by matching attributes across the sources (e.g., 'prev. close' vs 'Previous Close'), and then (2) reconciling daily

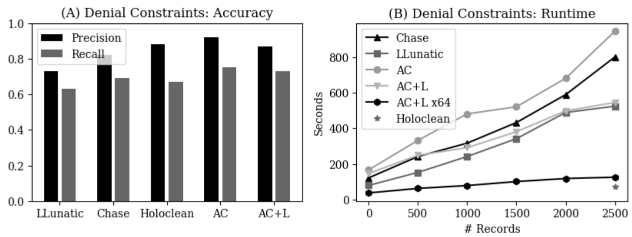


Figure 3: Comparison with denial constraint systems on the Flight dataset. (A) Arachnid (AC) matches or exceeds the accuracy of the specialized systems. (B) The structure of the data errors, along the learning and parallelization (AC+L x64) lets Arachnid scale sub-linearly and outperform all but HoloClean's reported runtimes.

ticker values values using primary-key-based functional dependencies akin to the flight dataset.

Terrorism: The Global Terrorism Database [16] 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 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.

7.2 Comparisons with Specialized Cleaning

This subsection compares Arachnid to specialized cleaning approaches. We illustrate that Arachnid can achieve comparable or higher accuracy, and the trade-offs in its runtime performance as compared to the baselines.

7.2.1 Denial Constraints

Denial constraints express a wide range of integrity constraints and form the basis of many data cleaning systems. Although Arachnid may not fully enforce integrity constraints, we can compose a quality function can quantifies the number of constraint violations and find transformation that reduce the number of violations. We use the Flight dataset for these experiments.

Baselines: We run against (1) Llunatic, a denial constraint-based cleaning system [13] implemented in C++ on top of PostgreSQL[1], and (2) a restricted chase algorithm [7] 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 [31] that used the same datasets and constraints, but did not run the experiment ourselves.

Results: Figure 3a 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 3b). 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.

[1]Constraints are specified as Tuple-Generating Dependencies

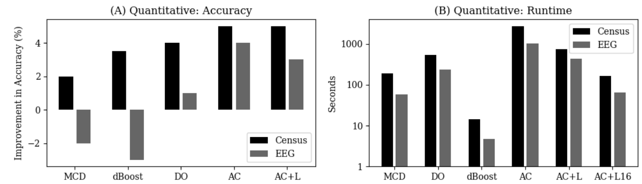


Figure 4: Quantitative data cleaning on census and EEG datasets for a classification application. Arachnid tranformation can clip outliers or set them to a default value. (A) Arachnid ha higher accuracy than outlier detection algorithms (MCD, dBoost), and Arachnid with a single transform template (DO). (B) Optimizations improve Arachnid runtime by over an order of magnitude.

7.2.2 Quantitative Data Cleaning

This experiment performs numerical cleaning on machine learning data. 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 [18]. 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 Cleaning (NC)*, *Minimum Covariance Determinant (MCD)* is a robust outlier detection algorithm used in [6] and sets all detected outliers to the mean value, *dBoost* uses a fast partitioned histogram technique to detect outliers [26], 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 run-

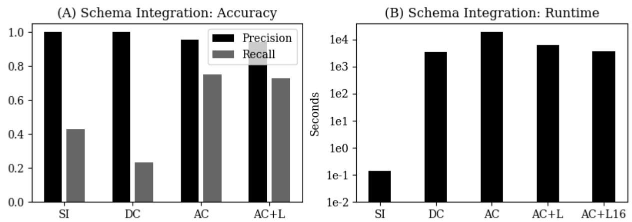


Figure 5: Schema integration and functional dependency errors in the stock dataset. Cleaning schema integration (SI) or functional dependencies (DC) in isolation results in high precision but poor recall. Arachnid cleans both types of errors with higher recall, and is as fast as DC.

times. dBoost is specialized for fast outlier detection and Arachnid is unlikely to match its runtime.

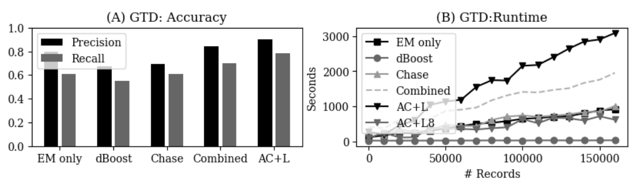


Figure 6: The Global Terrorism Database is a dataset of terrorist attacks scraped from news sources since 1970. (A) Shows that Arachnid can integrate many different forms of cleaning that were previously handled by disparate systems, (B) Arachnid achieves a competitive runtime to using all of the different system and accounting for data transfer time between them.

7.2.3 Schema Integration

We now evaluate cleaning on the stock dataset [14] from 55 different sources that contains a mixture of schema integration and functional dependency errors.

Baselines: We compare approaches for each error type in isolation. Schema Integration (SI) matches attributes based on attribute name string similarity, and Data Cleaning (DC) assumes schemas are consistent and enforces functions dependencies using a restricted chase [7].

Results: The specialized cleaning approaches exhibit high precision but low recall, as they miss records with multiple errors (Figure 5). Arachnid can mix both tasks in the quality function and has comparable precision with much higher recall. SI is significantly faster since it only reads schema metadata, however Arachnid with learning and 16 threads is competitive with DC.

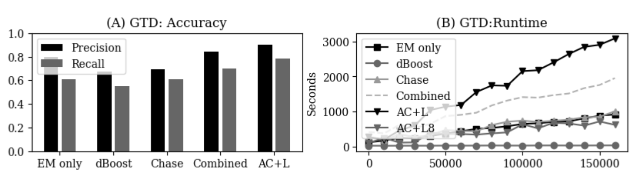


Figure 7: The terrorism dataset contains 3 classe of errors. (A) A unified cleaning framework outperforms individual cleaning approaches and even a serial combination. (B) Arachnid is easily parallelized to 8 threads to outperform the combined baselines.

7.3 Mixing Error Classes

This experiment, we use the multi-error Terrorism dataset [16].

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).

Results: Figure 7a shows that combining and cleaning 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 cleaning 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.

7.4 Arachnid In Depth

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

7.4.1 Algorithmic Sensitivity

Block-wise Cleaning: 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. Figure 8a 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: Figure 8b 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 cleaning 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 cleaning operation. Figure 8c 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 cleaning 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. Figure 8d shows that the F1 accuracy decreases as more constraints are added, however the F1 score is still above 75%.

7.4.2 Cleaning Generalization and Overfitting

An important characteristic of generating cleaning solutions as *programs* is that we can evaluate the program's robustness in terms

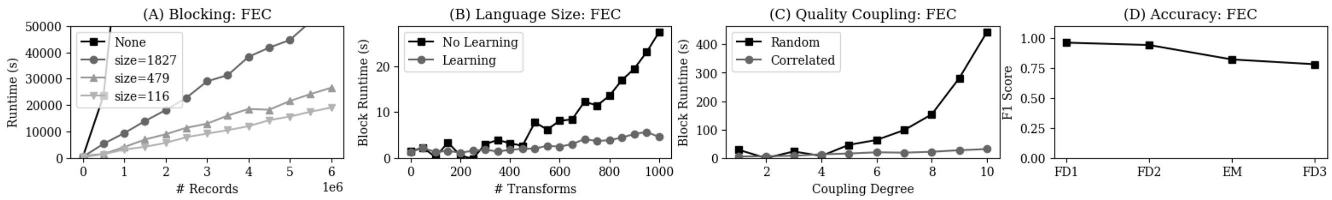


Figure 8: (A) The degree of block-wise partitioning directly affects search time, (B) increasing the transformation language Σ exponentially increases the search time, but learning is very effective at pruning the increased search space, (C) quality functions that couple errors between random records are significantly harder to optimize, (D) Arachnid degrades gracefully when adding irrelevant error types to the problem.

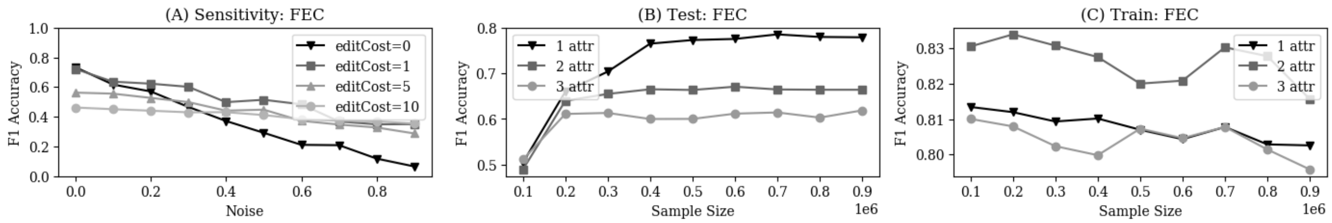


Figure 9: (A) Regularization by increasing an editCost penalty makes Arachnid more robust to noisy (mis-specified) quality functions, (B-C) overly expressive transformation templates can lead to overfitting (2 attr) or an infeasibly large search space (3 attr).

of machine learning concepts such as overfitting and generalization. To this end, we examine two concepts in the context of data cleaning: regularizaton and overfitting. We also find that Arachnid's high level interface is helpful for iteratively tuning the cleaning process.

Regularization: Misspecified quality functions can cause Arachnid to output poorly performing cleaning 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 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 cleaning programs.

Discussion: We have shown that data cleaning can overfit, and believe this is a potential issue in *any* cleaning procedure. These results highlight the importance of domain experts to judge and constrain the cleaning 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 cleaning language.

7.4.3 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 i^{th} 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.

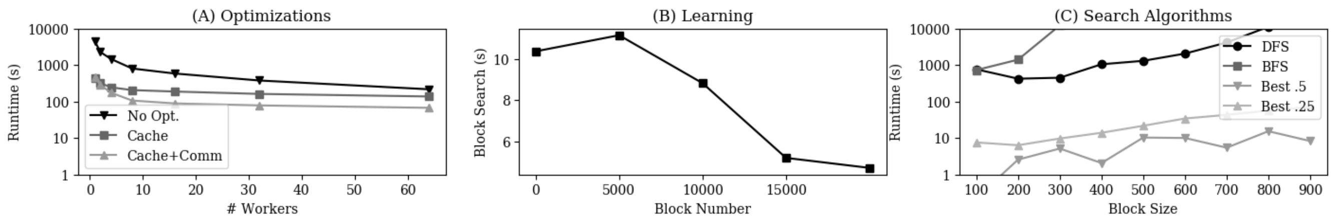


Figure 10: (A) Both the materialization (Cache) and distributed communication (Comm) optimizations contribute to improved scale-out runtimes. (B) The learned pruning rules improve the search costs for each subsequent block-wise partition. (C) Best-first search is better than BFS and DFS; reducing γ prunes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

7.4.4 Program Structure

Finally, we present results describing the structure of the data cleaning programs found with Arachnid. It is often the case that the program found by Arachnid is a concise description of the needed data cleaning 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 data cleaning 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. Contrary to prevailing wisdom, our experiments on 8 diverse datasets suggest that this general framework can still exploit problem-specific structure to achieve parity with specialized solutions in terms of cleaning accuracy, use pruning and parallelization to run comparably to specialized solutions, yet maintain the flexibility to clean datasets that span error class. Because the output is a cleaning program rather than a cleaned dataset, we are able to evaluate the cleaning results in terms of overfitting and generalization, and show how the high-level interface allows the user to easily control overfitting by tuning the quality function and language expressiveness.

Although our results suggest that borrowing from recent advances in planning and optimization is a fruitful direction, the results are counter-intuitive and raise a number of questions about future opportunities in data cleaning. Does Arachnid achieve comparable and higher accuracy than specialized systems because the systems are designed for worst-case scenarios that are not present in most datasets? Or are the benchmarks we borrowed and compared against too simple and amenable to greedy brute-force search? These are all possible explanations for our results, which are still somewhat counter-intuitive to us. We hope to really characterize and understand these tradeoffs in the future.

Finally, we are excited to extend Arachnid towards a more flexible and usable 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 (<200LOC 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] Ray: A high-performance distributed execution engine. https://github.com/ray-project/ray.
[3] Trifacta. https://www.trifacta.com/.
[4] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. In *VLDB*, 2014.
[5] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: a framework for integrating entity resolution with query processing. In *VLDB*. VLDB Endowment, 2015.
[6] 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.
[7] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, 2017.
[8] M. Bergman, T. Milo, S. Novgorodov, and W. C. Tan. Query-oriented data cleaning with oracles. In *SIGMOD*, 2015.
[9] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.
[10] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, 2014.
[11] Y. Chung, S. Krishnan, and T. Kraska. A data quality metric (dqm): How to estimate the number of undetected errors in data sets. 2014.
[12] E. F. Codd. A relational model of data for large shared data banks. In *Communications of the ACM*. ACM, 1970.
[13] 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.
[14] X. L. Dong. Data sets for data fusion experiments. http://lunadong.com/fusionDataSets.htm.
[15] W. Fan and F. Geerts. *Foundations of Data Quality Management*. 2012.
[16] N. C. for the Study of Terrorism and R. to Terrorism (START). Global terrorism database [data file]. https://www.start.umd.edu/gtd/, 2016.
[17] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *PVLDB*, 2001.
[18] 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.
[19] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. In *VLDB*, 2015.

[20] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.

[21] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.

[22] 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.

[23] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu. Towards reliable interactive data cleaning: A user survey and recommendations. In *HILDA*, 2016.

[24] 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.

[25] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. In *PVLDB*, 2016.

[26] Z. Mariet, R. Harding, S. Madden, et al. Outlier detection in heterogeneous datasets using automatic tuple expansion. 2016.

[27] 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.

[28] 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.

[29] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. In *IEEE Data Eng. Bull.*, 2000.

[30] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[31] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. In *arXiv*, 2017.

[32] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment*, 9(4):348–359, 2015.

[33] S. Russell, P. Norvig, and A. Intelligence. A modern approach. In *AI, Prentice-Hall*, 1995.

[34] B. Shneiderman. Dynamic queries, starfield displays, and the path to spotfire. 1999.

[35] 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.

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

[37] 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.

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

[39] 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.

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