Wisteria: Nurturing Scalable Data Cleaning Infrastructure

Daniel Haas, Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, Eugene Wu [†*]
UC Berkeley [†]Columbia University
{dhaas,sanjay,jnwang,franklin}@cs.berkeley.edu, ewu@cs.columbia.edu

ABSTRACT

Analysts report spending upwards of 80% of their time on problems in data cleaning. The data cleaning process is inherently iterative, with evolving cleaning workflows that start with basic exploratory data analysis on small samples of dirty data, then refine analysis with more sophisticated/expensive cleaning operators (i.e., crowdsourcing), and finally apply the insights to a full dataset. While an analyst often knows at a logical level what operations need to be done, they often have to manage a large search space of physical operators and parameters. We present Wisteria, a system designed to support the iterative development and optimization of data cleaning workflows, especially ones that utilize the crowd. Wisteria separates logical operations from physical implementations, and driven by analyst feedback, suggests optimizations and/or replacements to the analyst's choice of physical implementation. We highlight research challenges in sampling, in-flight operator replacement, and crowdsourcing. We overview the system architecture and these techniques, then propose a demonstration designed to showcase how Wisteria can improve iterative data analysis and cleaning. The code is available at: `http://www.sampleclean.org`.

1. INTRODUCTION

The prevalence of dirty data presents a fundamental obstacle to modern data-driven applications, since blindly using results that are derived from dirty data can lead to hidden, yet significant errors. Analysts report spending upwards of 80% of their time on problems in data cleaning [9] including error diagnosis, instance-specific cleaning scripts, and managing large data ingest pipelines. The underlying problem is that data cleaning is often specific to the domain, dataset, and eventual analysis. The analyst is faced with a breadth of possible errors that are manifest in the data and a variety of options to clean it. She must go through the cleaning process via trial and error, deciding for each of her data sources what to extract, how to clean it, and whether that cleaning will significantly change results.

Data cleaning is inherently iterative and Figure 1 shows a common progression for the development of a data cleaning plan, in this case the extraction of a restaurant's address from its unstructured webpage. While this operation can easily be represented at a *logical* level by its input and output schema, there is a huge space of possible *physical* implementations of the logical operator. For example, extraction could be rule-based, rely on structured learning, ask crowd workers to extract the desired data fields, or some combination of all three. Even after selecting (say) a crowd-based operator, many parameters might influence the quality of the output data or the speed and cost of cleaning it: the number of crowd workers who vote on the extraction for a given webpage, the amount each worker is paid, etc. A priori, a data analyst has little intuition for what physical plan will be optimal in this large space.

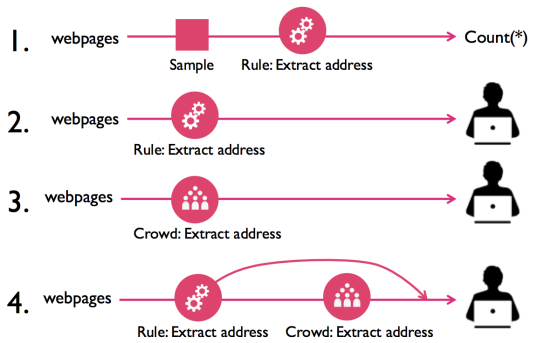

Figure 1: Example iterations on the design of the portion of a cleaning plan that extracts restaurant addresses from their unstructured webpages. 1) An exploratory plan that uses a sample to evaluate a simple address extraction method. 2) A plan that applies the method to the entire dataset. The quality is unsatisfactory. 3) An alternate plan that uses manual crowd extraction. The quality is now high, but the crowd-based extractor is slow. 4) A hybrid plan that sends only difficult webpages to the crowd, maximizing accuracy without sacrificing latency.

For the motivating example in Figure 1, the analyst will likely start by sampling the data, applying a simple method such as a rule-based approach, and querying the output to get a sense of the method's effectiveness (Figure 1.1). If the approach seems promising, she will apply the method to the entire dataset (Figure 1.2). However, the initially attempted approach seldom yields the desired accuracy, so the data analyst might have to experiment with other physical operators, for example a crowd operator that uses human workers to manually perform the extraction (Figure 1.3). Making the decision to switch to a crowd operator is complex as while the crowd might improve accuracy, it is more costly and time-consuming. Ideally, to achieve the desired accuracy without sacrificing speed, the data analyst use a hybrid between crowds and automated techniques (Figure 1.4). In the evolution of this data cleaning plan, our data analyst needs to make many decisions about the choice of physical operators by reasoning about their latency, accuracy, and cost. Without a general, scalable, and interactive system

[*]Work conducted while visiting UC Berkeley

that supports rapid iteration on candidate cleaning plans, analysts have to manually construct a physical plan and making the wrong decision, for example using the crowd when it only marginally improves accuracy, can be very costly.

However, no existing systems address the end-to-end iterative data cleaning process described above. Extract-transform-load (ETL) systems [1–3] require developers to manually write data cleaning rules and execute them as long batch jobs, and constraint-driven tools allow analysts to define "data quality rules" and automatically propose corrections to maximally satisfy these rules [6]. Unfortunately, neither provide the opportunity for iteration or user feedback, inhibiting the user's ability to rapidly prototype different data cleaning solutions. Projects such as Wrangler [4,8] and OpenRefine [14] support iteration with spreadsheet-style interfaces that enable the user to compose data cleaning sequences by directly manipulating a sample of the data and applying these sequences to the full dataset. However, they are limited to specific cleaning tasks such as simple text transformations, do not support crowd-based processing at scale, and cannot incorporate user feedback to optimize the physical implementation of the data cleaning sequences. Crowd-based [7,12] systems have been proposed to relieve the data cleaning analyst of the burden of rule specification or manual cleaning, but are usually very task specific (e.g., deduplication [5,7,10,11]) and require analysts to awkwardly chain systems together to execute the entire cleaning workflow, preventing end-to-end optimization of the entire plan. These existing limitations suggest the need for a system that is general enough to adapt to a wide range of data cleaning applications, scales to large datasets, and natively supports fast-feedback interactions to enable rapid data cleaning iteration.

In this paper, we introduce Wisteria, a system designed to support the iterative development and optimization of data cleaning plans end to end. Wisteria allows users to specify declarative data cleaning plans composed of rule-based, learning-based, or crowd-based operators, then enabling rapid iteration on plans with cost-aware recommendations for improving the accuracy or latency of a plan by swapping in new physical operators or modifying their parameters. Early, exploratory plans are supported with sampling and approximate query processing techniques [15]. Wisteria additionally provides users with the opportunity to provide ground-truth feedback after each logical operator in the plan to improve recommendations, and supports adjusting in-flight plans efficiently using caching and tuple lineage.

Supporting these capabilities requires a combination of careful engineering as well as tackling several research challenges:

- **Sampling**: We provide sampling as a first-class logical operator for data cleaning plans that tolerate approximation, and use it to speed up iteration on early-stage plans.
- **Recommendation**: We leverage interactive user feedback to recommend cost-aware changes to in-flight cleaning plans that allow users to trade off accuracy and latency, and provide efficient mechanisms for implementing recommended changes without re-executing the plan on already cleaned tuples.
- **Crowd Latency**: We leverage techniques for straggler mitigation [13] and model crowd worker speed and accuracy to reduce the (often rate-limiting) latency of crowd data cleaning, consistently retrieving results in seconds rather than hours.

In our demonstration, we will run an entity resolution plan on two restaurant datasets, and show how Wisteria can be used to 1) specify and execute a data cleaning plan using our domain specific language, 2) quickly clean a sample to characterize how a plan is performing, 3) observe that cleaning plans are not necessarily optimal across datasets, and 4) incrementally refine the plan to fit a new dataset by changing an operator's parameter (e.g., similarity threshold) or its physical implementation. The fourth interaction lets the participants inspect the effects of different physical plans. Users can then execute a plan over a live crowd that uses the audience as workers, or a simulated crowd that uses pre-collected crowd responses. The dashboard (Figure 4) also provides a live inspection interface to view the status of the cleaning plan as it executes.

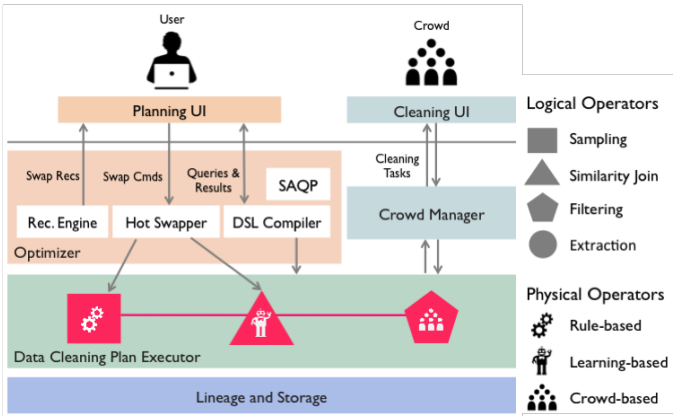

Figure 2: Wisteria system architecture, with an example entity resolution plan.

2. SYSTEM ARCHITECTURE

In this section, we provide a brief overview of the Wisteria system and its APIs. Figure 2 depicts the system architecture.

2.1 Architecture Overview

The Wisteria architecture provides UI, language, and systems tools for building data cleaning plans. Users interact with the system through the **Planning UI**, which allows them to compose data cleaning workflows from modular operators. These workflows are represented as expressions in our data cleaning language (section 2.2), then synthesized as data cleaning plans by our **DSL compiler**. As the **Data Cleaning Plan Executor** executes the compiled plans, users can interact with the plans via tight feedback loops in two ways. First, users can issue queries to the **SAQP module** and observe approximate results based on the data that has been cleaned thus far. Second, the **Recommendation Engine** displays a set of suggested modifications to the active cleaning plan (for example, making a similarity join more permissive) in the **Planning UI**, and users can update the data cleaning plan in-flight by accepting a suggestion and using the **Hot Swapper** to modify components of the pipeline. Intermediate results and cleaned data are maintained in a **Lineage and Storage engine** that tracks each tuple's lineage in order to enforce the semantics of hot-swapping correctly on in-flight tuples. Logical cleaning operators may have a number of physical implementations (section 2.3). Automated rule-based or learning-based operators leverage Spark and MLLib for efficient distributed computation, and operators that require human intervention call out to Wisteria's **Crowd Manager** API, which renders data cleaning tasks and displays them to crowd workers from multiple crowds (e.g., Amazon Mechanical Turk) in a web-based **Cleaning UI** for processing.

2.2 Cleaning DSL

We provide a language for specifying the composition of data cleaning operators. The logical operators define the input and output behavior of the operation and the physical operators specify the implementation. The general syntax of this language is:

```
<logical operator> on <relations>
```

```
with <physical operators> , <params>
```

These expressions are composable. For example, the following represents the data cleaning plan in Figure 2 (an entity resolution plan):

```
Filtering on (
    SimilartyJoin on (
        Sampling on BaseTable
        with Uniform)
    with Jaccard, thresh=0.8)
with CrowdDeduplication, numVotes=3
```

Additionally, Wisteria provides integration of our DSL with Scala/Apache Spark, allowing SchemaRDDs (Spark RDDs with additional schema information) to serve as base tables in expressions.

2.3 Cleaning Operators

Wisteria supports a small set of operators that can express a wide variety of common data cleaning workflows. For example, the pipeline depicted in Figure 2 performs crowd-based entity resolution: the similarity join operator generates candidate tuple pairs (the *blocking* step), and the crowd-based filter operator uses humans to identify duplicates from the candidates (the *matching* step). Additional operators include Extraction and Sampling.

Individual logical operators have multiple physical implementations, each with its own cost, latency, and accuracy profile. For example, crowd-based implementations tend to be high cost, high latency, and high accuracy, whereas rule-based implementations tend to be low cost, low latency, and low accuracy. The `with` clause of our data cleaning language allows users to explicitly specify desired physical operators, and Wisteria's recommendation engine provides actionable suggestions for modifying the pipeline to navigate the tradeoff space.

3. RESEARCH CHALLENGES

To support evolving data quality needs, there are three main research challenges in Wisteria: (1) sampling, (2) recommendation, and (3) crowd sourcing.

3.1 Sampling

In prior work, we explored the problem of estimating aggregate query results over dirty data [15]. In SampleClean [15], we found that aggregate queries can often be answered with very high accuracy (i.e 99%) with only a small fraction of clean data, and we can clean just enough for the application's data quality requirements. In the context of the iterative design, data analysts often run aggregate queries, e.g., count the number of Chinese restaurants, on a new data source to assess the quality of the data. In Wisteria, we implement sampling as a logical operator that can be used for quickly prototyping and optimizing workflows on samples of data and then transferring these optimizations to full datasets. There are many additional research opportunities with sampling. For example, sampling allows for quick introspection of otherwise opaque operators, such as testing the quality of a crowd with a small set of records. We also build a significance testing framework to allow users to declare aggregate queries of interest and notify them when a change to a plan has a statistically significant effect.

3.2 Recommendation

Our next research challenge is to recommend changes to a data cleaning plan based on user feedback. The user can inspect the output of an operator and identify result tuples that are incorrect. This feedback is operator specific. For example, in a Similarity Join the user can mark false positives (matched pairs that should not be similar) and false negatives (unmatched similar pairs) and in an Extraction the user can mark incorrect attribute values. We highlight two key challenges: generating recommendations and applying the recommendations mid-execution.

Recommendations: There are three types of recommendations: (1) parameter change, (2) operator replacement, and (3) operator addition.

Parameter Change. Many of the physical operators in Wisteria have tunable parameters, whose values is often very dataset-specific, and the user feedback gives us a way to evaluate the quality of the initial parameter choice. For example, Similarity Joins have a similarity threshold and a similarity function. Increasing this threshold reduces the selectivity of the join, and Wisteria chooses a threshold that maximizes the F1-score.

Operator Replacement. Wisteria recommends changes to these physical operators when the user indicates that they are not satisfied with the output. For example, we can use the user feedback as training examples in our learners as an estimate of the crowd performance. This allows us to estimate the value of replacing the physical operator with an active learning variant. Additionally, we can try different variants of automated operators to test how accurate they are with respect to the user feedback.

Operator Addition. There are also cases where we may want to add another physical operator, while still preserving the logical input-output behavior of the workflow. It is common in extraction tasks to have most records accurately extracted with an automated extractor but only a small subset requiring additional inspection. For these cases, we can add a crowd-based Filter operator to separate these examples for additional cleaning.

Dynamic Modification: To be able to quickly modify cleaning plans after recommendations, we explore ways to intelligently reuse computation. Because cleaning can be time-consuming, it is inefficient to restart the plan every time the user wants to make a small change. We design a framework for *hot swapping* plan operators that re-uses existing results while ensuring that the output of the swapped plan is the same as if the new plan had been run from the start.

Caching allows for result re-use if a downstream operator is modified or added. If the system has sufficient memory, then we can cache all of the intermediate results. However this is not always possible, and the key challenge is to select which results to cache. To do this, we have to integrate the caching framework with our recommendation engine. When we make a recommendation for a change, we must cache the preceding operator.

Lineage allows us to understand how results change if upstream operators are modified. For example, decreasing a similarity join threshold increases the number of output pairs, but adding an additional filtering step reduces the number of output tuples. The key property here is monotonicity, and some types of monotone Filter and SimilarityJoin are data cleaning analogs for a Select-Join relational algebra. We can therefore model upstream hot-swapping as an incremental view maintenance problem and update the final result based on the insertion or deletion of tuples earlier in the plan.

3.3 Crowdsourcing

Working with crowds is inherently challenging: unlike with automated operators, the accuracy and speed of processing each tuple varies widely with the crowd worker assigned to it. Completion time of an operator depends on the response times of individual workers, and on real-world crowdsourcing platforms, the distribution of response latencies is highly skewed; analogous to the straggler problem in distributed systems. We address this problem by maintaining a pool of high-speed, high-quality crowd workers

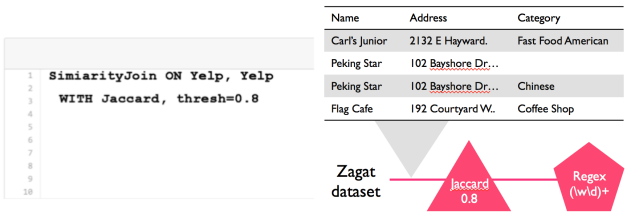

Figure 3: The dashboard contains both a visual interface and a text box to specify data cleaning operations. When the user is satisfied, she can run the plan and see the results on the right.

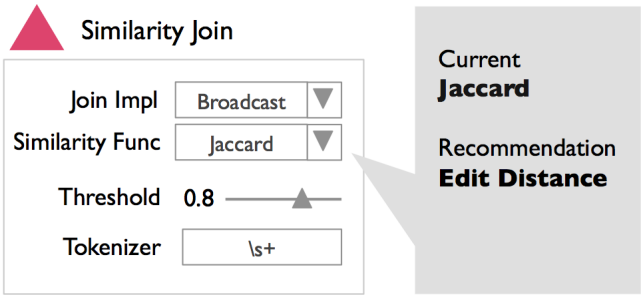

Figure 4: The operator view lists the parameters of an operator. Users can view recommended changes and modify parameters on the fly.

and develop task routing strategies that can avoid assigning tasks to slow workers and leverage redundancy to significantly reduce the time that is required to clean data with the crowd. Additionally, active learning techniques reduce the number of tuples that require crowd work to clean the data. Another challenge with crowd operators is that some workers may give inccorect responses. We modify state-of-the-art quality control techniques for the active learning setting using redundancy and Expectation-Maximization based voting algorithms.

4. DEMONSTRATION

In this section, we detail the proposed demonstration. The objective of this demonstration is to illustrate how Wisteria enables the rapid iterative construction of data cleaning plans and the ability to transfer workflows between similar dirty datasets.

4.1 Datasets

In our demo, we will consider entity resolution tasks on two different restaurant datasets. The first dataset contains 858 Zagat reviews[1], each tagged with the cuisine of the restaurant reviewed (e.g. “Chinese" or “French"). The second dataset is from Yelp and contains 58,127 restaurant records that are also tagged with a category. In both datasets, tags are inconsistent across records, e.g. “Chinese" vs. “Chinese Cuisine". We will use Wisteria to merge similar categories together and find the top 10 most popular categories in the dataset.

4.2 Demo Walkthrough

Now, we will detail the steps of the proposed demonstration. A screenshot of our dashboard interface is illustrated in Figure 3.

Step 1: We will explain the components of our dashboard interface to the demo participants, including how to design data cleaning plans with the visual drop down menus, how to compile those plans to our DSL, and how to evaluate the results.

Step 2: Our dashboard will be pre-populated with a data cleaning plan for tag deduplication, and will start off with the Zagat restaurant dataset. Participants will have the option of choosing one of two Similarity Join implementations, Edit Distance and Jaccard Similarity, and can tune the thresholds for either. Participants can also add a crowdsourced filtering step in addition to similarity thresholding.

Step 3: When a participant is satisfied with a plan, they can hit “Approve" to execute the data cleaning. If they chose to use crowdsourcing, then they can complete crowd tasks. The results of the plan are visualized in the upper right of our interface. We show a representative sample of changed records allowing the participant to understand how cleaning affects the data. Participants can further analyze their plan by clicking on an operator (Figure 4). In addition to the parameters, this shows the recommended changes to the operator. For example, Figure 4, shows a recommendation to change the similarity metric from Jaccard to Edit Distance since the attribute in question does not have many tokens.

Step 4: Participants can then switch datasets and observe how the same plan performs on another dataset. They can modify the plan using the visual interface until the results are satisfactory.

[1] www.cs.utexas.edu/users/ml/riddle/data/restaurant.tar.gz

5. REFERENCES

[1] Apache falcon. http://falcon.apache.org.

[2] Informatica. https://www.informatica.com.

[3] Talend. https://www.talend.com/solutions/etl-analytics.

[4] Trifacta. http://www.trifacta.com.

[5] Z. Chen and M. Cafarella. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1126–1135. ACM, 2014.

[6] 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 Conference*, pages 541–552, 2013.

[7] 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.

[8] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[9] S. Kandel, A. Paepcke, J. Hellerstein, and H. Jeffrey. Enterprise data analysis and visualization: An interview study. *VAST*, 2012.

[10] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[11] H. Park and J. Widom. Crowdfill: Collecting structured data from the crowd. In *SIGMOD*, 2014.

[12] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[13] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 301–316. USENIX Association, 2014.

[14] R. Verborgh and M. De Wilde. *Using OpenRefine*. Packt Publishing Ltd, 2013.

[15] 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 Conference*, pages 469–480, 2014.