The Data Civilizer System

ABSTRACT

In many organizations, it is often challenging for users to find relevant data for specific tasks, since the data is often scattered across the enterprise and often inconsistent. In fact, data scientists routinely report that the majority of their effort is spent finding, cleaning, integrating, and accessing data of interest to a task at hand. In order to decrease the "grunt work" needed to facilitate the analysis of data "in the wild", we present DATA CIVILIZER, an end-to-end big data management system. DATA CIVILIZER has a *linkage graph computation* module to build a linkage graph for the data and a *data discovery* module which utilizes the linkage graph to help identify data that is relevant to user tasks. It also leverages the linkage graph to link the discovered data, so it can be queried, along with a polystore DBMS for query execution, which also integrates data cleaning operations.

In practice, different tasks might invoke the above modules in different orders, and might be iterative. To cope with this, the system has a *workflow* engine to enable the arbitrary composition of different modules, as well as to handle data updates. We have deployed our preliminary DATA CIVILIZER system in two institutions, MIT and Merck. Users have given us positive feedback and indicated that the system indeed shortened their time and effort to find, prepare, and analyze the data.

1. INTRODUCTION

An oft-cited statistic is that data scientists spend 80% of their time finding, preparing, integrating, and cleaning data sets. The remaining 20% is spent doing the desired analytic tasks. In practice, 80% may be a lower bound; for example one data officer, Mark Schreiber of Merck, a large pharmaceutical company, estimates that data scientists in Merck spend 98% of their time on "grunt work" and only one hour per week on "useful work".

In this paper, we present DATA CIVILIZER, a system we are building at MIT, QCRI, Waterloo, and TU Berlin, whose main purpose is to decrease the "grunt work factor" by helping data scientists to (i) quickly *discover* data sets of interest from large numbers of tables; (ii) *link* relevant data sets; (iii) *compute* answers from the disparate data stores that host the discovered data sets; (iv) *clean* the desired data; and (v) *iterate* through these tasks using a workflow system, since data scientists often perform these tasks in different orders.

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

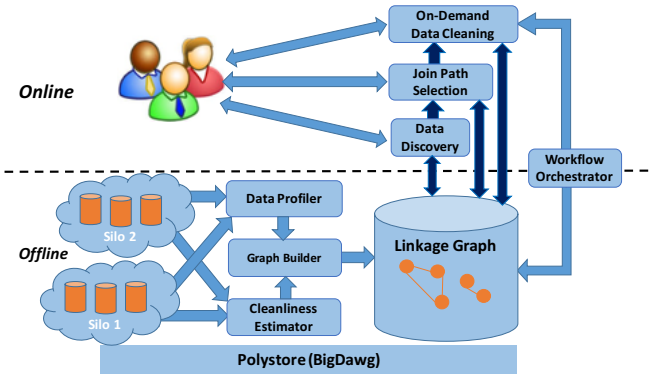

Figure 1: Data Civilizer Architecture

DATA CIVILIZER consists of two major components, as shown in Figure 1. The *offline component* indexes and profiles data sets; these profiles are stored in a linkage graph, which is used to process workflow queries online. Data sets in DATA CIVILIZER consist of structured data, which may be stored in relational databases, spreadsheets, or other structured formats. In the remainder of the paper, we use the term *tables* or *data sets* to refer to these. The *online component* involves executing a user-supplied workflow that consists of a mix of discovery, join path selection, and cleaning operations on data sets, all supported via interactions with the linkage graph. We elaborate on each module in the remainder of this section, using Merck as an example.

[Linkage Graph Computation.] The linkages amongst attributes and tables are needed so that the user can discover interesting data, assemble relational schemas, and run SQL queries on the discovered data sets. Initially, all the linkages that can be found in linear time are built during the offline stage. As existing data changes and new tables are added, the graph is incrementally updated to reflect the changes. All the other linkages, such as primary key-foreign key (PK-FK) relationships, will be built in the background or on-demand during the online stage. Our current prototype identifies PK-FK relationships, using inclusion dependencies. As such, the process of building the linkage graph can be run dynamically and on-demand; we discuss its details in Section 2.

[Discovery.] A data scientist at Merck has a hypothesis, for example, *the drug Ritalin causes brain cancer in rats weighing more than 300 grams*. His first job is to identify relevant data sets, both inside and outside of Merck, that might contribute to testing this hypothesis. Inside the company alone, Merck has approximately 4,000 Oracle databases and countless other repositories. The discovery

component in DATA CIVILIZER will assist the scientist in finding tables of interest from all the Merck tables. Discovery queries are run as a part of the online workflow which is discussed in Section 3.

[Polystore Query Processing and Curation.] Since organizations such as Merck have a variety of massive-scale data storage systems, it is not feasible to move all data to a central data warehouse. Also, it is neither economically nor technically practical to perform data processing on all of the thousands of databases in advance. DATA CIVILIZER is built using a polystore architecture [11] that federates query processing across disparate systems inside an enterprise. Our plan is to leverage the `BigDAWG` polystore system [12] to pull data from multiple underlying storage engines to compute the final result (or the *view*) that satisfies the user's specifications.

Obviously, data cleaning, data transformation and entity consolidation must be integrated with querying the polystore and constructing the desired user view. This is an expensive process with the human effort required to validate cleaning decisions being the most important cost. Since there may be multiple views that "solve" a data scientist's query, each with different accuracy and human validation cost, DATA CIVILIZER must estimate the cost of curating the possible views, given a scientist's time budget. The above aspects will be discussed in Section 4.

[Updates.] If a source data set is updated, these updates must be incrementally propagated through the data curation pipeline to update downstream materialized views. In some cases, the human effort involved may be daunting and the materialized view should be discarded rather than updated. In addition, if a scientist updates a view, we need to propagate changes to other derived views, as well as back upstream to data sources, if this is possible. Section 5 discusses these issues.

[Workflow.] DATA CIVILIZER offers a workflow engine whereby data scientists can iterate over its components in whatever order they wish. Moreover, they need to be able to undo previous workflow steps and perform alternate branching from the result of a workflow step. Section 6 discusses our workflow management ideas.

We describe the state of the current implementation of DATA CIVILIZER in Section 7. We also report on initial user experience for two use cases: the MIT data warehouse and Merck. We conclude with final remarks and an outline of our future research plans in Section 8.

The main contribution of DATA CIVILIZER is that it is an "end-to-end" system. In contrast, there has been much work on "point solutions" that solve small pieces of the overall problem. For example, Data Wrangler [15] and DataXFormer [7] automate some aspects of cleaning and transforming data, Data Tamer [20] mainly performs schema mappings and record linkage, and DeepDive [19] extracts facts and structured information from large corpora of text, images and other unstructured sources. However, no solution performs discovery, linkage graph computation, and polystore operations in concert. In addition, we have recently studied several representative data cleaning systems on a collection of real world data sets"from the wild" [5] and we found out that there was no cleaning Esperanto. Hence multiple tools are required to achieve reasonable accuracy and point solutions neither offer enough functionalities nor achieve acceptable performance. Therefore an ensemble approach is required.

2. LINKAGE GRAPH COMPUTATION

In this section, we discuss how to build the linkage graph of DATA CIVILIZER. We first show how to create the data profiles in linear time—which can then be fed into the linkage graph—and then formally define our graph model in Section 2.2. Finally, as an example of linkages, we focus on how we discover and refine PK-FK relationships in Sections 2.3 and 2.4, respectively.

2.1 Data Profiling at Scale

The key idea is to summarize each column of each table into a *profile*. A profile consists of one or more *signatures*; each signature summarizes the original contents of the column into a domain-dependent, compact representation of the original data. By default, signatures for numerical values consist of a histogram representing the data distribution and; for textual values they consist of a vector of the most significant words in the column, indexed using a hash table. Our profiles also contain information about data cardinality, data type, and numerical ranges if applicable [6]. In addition to the profiles, we maintain a global keyword index to facilitate keyword queries. The index maps keywords to columns containing the word in their names or content.

To run at scale, data profiling relies on sampling to estimate the cardinality of columns and the quantiles for numerical data. This avoids having to sort each column, which is not a linear computation. This also allows us to avoid making a copy of the data, reducing memory pressure.

The profiler consists of a pipeline of stages. The first stage performs basic de-noising of the data, such as removing empty values and dealing with potential formatting issues of the underlying files, e.g., figuring out the separator for a CSV file. The second stage determines the data type of each column, information that is propagated along with the data to the next stages. The rest of the stages are responsible for computing cardinalities, and performing quantile estimation, e.g., to determine numerical ranges.

The profiler has connectors to read data from different data sources, such as HDFS files, RDBMS or CSV files. Data sets are streamed through the profiler pipeline; the pipeline outputs the results to a *profile store* where it is later consumed by the graph builder, which will be explained in Section 2.2. The profiler works in a distributed environment using a lightweight coordinator that splits the work among multiple processors, helping to scale the computation.

2.2 Graph Builder

The linkage graph is a graph with both simple nodes, which represent columns, and hyper-nodes which are multiple simple nodes that represent tables or compound keys. Each edge represents a relationship between two nodes (either simple nodes or hyper-nodes) in the graph. Note that in this way the model can express relationships involving individual columns as well as groups of them. This is necessary to capture relationships among tables, e.g., PK-FK relationship between compound keys.

Examples of relationships are *column similarity*, *schema similarity*, *structure similarity*, *inclusion dependency*, *PK-FK relationship*, and *table subsumption*. The node label contains table metadata, such as table name, cardinality and other information computed by the profiler. The edge label includes metadata about the relationship it represents, e.g., type and score, if any. Computing the different relationships requires different time complexities. We categorize them into *light relationships*, which can be computed in sub-quadratic time, and *heavy relationships*, which need at least quadratic time. The light relationships contain column similarity and schema similarity. The heavy relationships include primary key-foreign key (PK-FK), inclusion dependency, and structure similarity.

The graph builder first computes the light relationships amongst pairs of nodes in advance during offline processing. In particular, the graph builder first finds *column similarity* and *schema similarity*

relationships. Each edge is assigned a weight that represents the strength of the relationship: the particular meaning depends on the edge semantics. Also, edges with a similarity less than a user-defined threshold are discarded to avoid having a graph with too many edges that are not significant. A straightforward computation of such relationships requires a $O(n^2)$ computation, which will not scale. However, for light-relationships, which depend on similarity metrics such as edit distance and Jaccard similarlity, we can employ locality sensitive hashing (LSH) [10] to yield sub-quadratic runtime; we have found LSH to run quite well in practice, even for data sets up to several Terabytes.

It is usually expensive to compute the heavy relationships. To address this issue, the graph builder adopts the following general approach: (i) the computation of heavy relationships is performed in the background, (ii) data profiles and light relationships are used as much as possible to prune the search space for the heavy relationships, and (iii) after the user chooses the data of interest, if the heavy relationships are not ready amongst this data, the graph builder can construct the heavy relationships online.

We use the PK-FK relationships as an example of how we efficiently compute a heavy relationship and deal with the complexities that arise in real data where errors and noise make finding such relationships tricky. The graph builder utilizes an implementation of inclusion dependency (i.e., column A *includes all the values of* column B) to find candidate PK-FK relationships, since a PK should include all values in an FK column. We then refine the candidates using machine learning methods. However, as it is well known, data in the wild is quite dirty, which makes discovering inclusion dependencies difficult. Specifically, foreign keys may not match a primary key, because of errors in either the PK or the FK. To tolerate errors, we extend traditional inclusion dependency discovery by both key coverage and text similarity and propose an *error-robust inclusion dependency* approach. We describe this approach in Section 2.3 and the machine learning techniques to refine the candidate PK-FK relationships in Section 2.4.

2.3 Error-Robust Inclusion Dependency

As noted above, PK-FK relationships are usually identified using inclusion dependency techniques. To overcome the presence of dirty data, we propose an error-robust inclusion dependency scheme. Consider two columns (or compound columns) from two tables R and S denoted by $R[X]$ and $S[Y]$. If there is a foreign key constraint on $R[X]$ with reference $S[Y]$, all the values in $R[X]$ must appear in $S[Y]$, which yields an inclusion dependency from $R[X]$ to $S[Y]$. However, in the real world, values in foreign key fields may not exist in the primary key fields due to errors. In this case an inclusion dependency from foreign key to primary key does no hold.

To address this, for each distinct value in $R[X]$, we calculate the text similarity to values in $S[Y]$ and use the maximum value as the strength of a value matching. We then compute the total strength of the maximum matching value divided by the number of values in $R[X]$. This is the overall strength of the inclusion dependency. If this number exceeds a predefined threshold δ, we add an inclusion dependency from $R[X]$ to $S[Y]$ in our linkage graph.

When dealing with compound columns, we can utilize different text similarity functions on different fields. Also, we must compose the individual column scores to achieve an overall strength.

2.4 Candidate PK-FK Relationships Refinement

The error-robust inclusion dependency algorithm returns a collection of candidate PK-FK relationships. We use the algorithms in the work by Rostin et al. [18] to refine our candidate selection. The authors proposed 10 different features to distinguish foreign key constraints from spurious inclusion dependencies. Consider two columns $R[X]$ and $S[Y]$ with an inclusion dependency from the first one to the second one. The specified features include *coverage*, the ratio of distinct values in $R[X]$ that are contained in $S[Y]$, *column name similarity*, the similarity between the attribute names, and *out-of-domain range*, the percentage of values in $S[Y]$ not within $[\min(R[X]), \max(R[X])]$ where $\min(R[X])$ and $\max(R[X])$ are respectively the minimum and maximum values in $R[X]$.

Using the above features, we implemented the four machine learning classification algorithms from Rostin et al. [18]. These allow us to distinguish spurious inclusion dependencies from real ones with high accuracy. We add the highest confidence candidate PK-FK pairs as edges in the linkage graph.

Next we discuss how to utilize the linkage graph to help the user discover his interesting data in Section 3. The PK-FK relationships can be used to link the tables of user interest from the discovery module. However, there may exist multiple subgraphs in the linkage graph that can connect all the interesting tables, which we call *join paths*. In Section 4, we discuss the choice of which one(s) to use to materialize a view for the user.

3. DISCOVERY

The data discovery module's goal is to find relevant data from perhaps hundreds to thousands of data sets spread across the different storage systems of an enterprise, using the linkage graph to provide answers efficiently.

Current solutions to finding relevant data include asking an expert (if one is available) or performing manual exploration by inspecting data sets one by one. Obviously, both approaches are time-consuming and prone to missing relevant data sources. Recently, there have been efforts to automate this process, some examples are Goods [13] and InfoGather [21]. Goods permits users to inspect datasets and find others similar to datasets of interest. Infogather permits to extend the attributes of a dataset of interest with attributes from other datasets. These systems are designed to solve these particular use cases, and so their indexes are built for this specific purpose. In contrast, our discovery module aims to support general *discovery queries* that can then be used for unanticipated discovery needs. Hence, the discovery module uses a linkage graph (Section 2.2) to permit a broader range of queries.

The data discovery module narrows down the search for relevant data from the entire space of available data sources to a handful, on which finer-grained relationships can be computed efficiently. DATA CIVILIZER discovery supports a user-level API to search for relevant data using a variety of techniques such as searching for tables with specific substrings in the schemas (*schema search*) or table values (*content search*).

3.1 Discovery Queries

Users can submit *discovery queries* to find data sets whose schemas are relevant to the task at hand. For example, to find all data sets that contain names of employees in a company, the user issues a discovery query that searches for all attributes referring to an employee or a name. The user can then write a SQL query to filter from a list. Here are some examples of functions that the discovery module provides to users; each of these queries can be answered using data collected as a part of profiling as well as the information expressed in the linkage graph:

- **Fill-in schema**. Given a set of names, this function retrieves tables or groups of tables that contain the desired attributes. The core of the operation consists of finding tables with similar attribute names to provided ones, using the keyword index.

- **Extend attribute.** This is similar to the extend operator of Octopus [8], or the ABA operation of Infogather [21]. Using this function users can extend tables of interest with additional attributes. The core of the operation consists of finding matches to the current table using the linkage graph, and then retrieving attributes that do not appear.

- **Subsumption relationships.** Given some reference table, this function provides a list of tables or groups of tables that have some form of subsumption relationship (i.e., are contained in or contain) with respect to the reference table; this can be done directly from the linkage graph.

- **Similarity and lookup.** Discovery can also be used to find schemas that contain data similar to some provided schema (attribute or table), as well as to find schemas that contain certain values, i.e., keyword search for textual data and range search for numerical data.

Discovery works similarly to an information retrieval system: schema retrieval queries do not return a specific answer, but instead return a ranked set of results. We are currently exploring several ranking options, and a model that decouples discovery query processing from ranking evaluation seems the most promising.

4. POLYSTORE QUERY PROCESSING

DATA CIVILIZER uses the BigDAWG polystore [12] to federate access to multiple storage systems inside an organization. BigDAWG consists of a middleware query optimizer and executor, and shims to various local storage systems. We assume that a user has run discovery and that the corresponding linkage graph has been computed. If the user is interested in a composite table containing a particular set of columns that are not available in any single data source, we can use the PK-FK mining techniques described in the previous section to find a set of possible join paths that can be used to materialize the table of interest to the user. Furthermore, for any particular join path, it is straightforward to construct the BigDAWG query that materializes the view specified by each join path. However, we still need to choose which of several possible join paths is the best to compute the table of interest to the user.

4.1 Materialized View Selection

The conventional data federation wisdom is to choose the join path that minimizes the query processing cost. However, this ignores data cleaning issues, to which we now turn. To achieve high quality results, one has to clean the data in conjunction with querying the table. For example, if one has a data value, `New Yark`, and wants to transform it to one of its airport codes (`JFK, LGA`), then one must correct the data to `New York`, prior to the airport code lookup. Obviously, cleaning usually entails a human specification of the corrected value or a review of the value produced by an automatic algorithm. Such human costs can easily dominate the total query execution time and represent a significant financial cost in many data cleaning scenarios. As such, one goal of DATA CIVILIZER is to choose the join path that produces the highest quality answer, rather than the one that is easiest to compute.

In DATA CIVILIZER, a user has to decide how to trade off data quality and cleaning cost. DATA CIVILIZER defines two parameters, both under the user's control.

1. Minimize cost for a specific cleanliness metric. In this case, the user requires the data to be a certain percentage, P, correct and will spend whatever it takes to get to that point.
2. Maximize accuracy for a specific cost. In this case, the user is willing to spend M and wishes to make the data as clean as possible.

Sometimes the user is the one actually cleaning the data. In this case, (s)he can use P and M to quantify the value of her/his time. In other cases, cleaning is performed by other domain experts, who generally need to be paid. In this case, P and M are statements about budget priorities.

Sometimes the user may prefer the join path that yields largest view size, i.e., the number of rows in the view. For example, the user may want a view with 1,000 rows and 90% cleanliness rather than a view with 10 rows and 98% cleanliness. We can combine all the facets as the criteria for join path selection. Estimating result set sizes has been extensively studied in the literature [14] and we just utilize the cardinality estimates produced by BigDAWG.

As a result, DATA CIVILIZER must make the following decisions. First, it needs to assess the cleanliness of the result of any given join path. We treat this issue in Section 4.2. Then, we need to choose where to place, in the resulting query plan, data cleaning operations to be the most efficient. This is the topic of Section 4.3.

4.2 Cleanliness Model

In DATA CIVILIZER, we collect information about the errors in each data set. Specifically, we need an accuracy metric for each column, namely an estimate for the percentage of the column which is erroneous. There are multiple ways to achieve this goal including: (i) Have the DBA who added the table make an estimate for the accuracy of each column. (ii) Estimate the number of null values and outliers in the column, and assume that a second accuracy estimator is the inverse of this number. (iii) Conduct a similarity join on any foreign key and the primary key and assume any mismatched values are errors. We can also assume that these error estimates can be used for the non-key fields in the table. (iv) Cluster the tables with similar primary keys and assume that their records have the same semantic information. Therefore, we identify the columns in the cluster which are similar, and use majority voting to estimate the cleanliness of each column. All of these methods can generate the error estimates we need. Hence, we can linearly combine these estimates based on our confidence in the four methods. For example, we can assign a high coefficient to the first method if the DBA is trustworthy. This gives us a mechanism to assign a cleanliness estimate to any non-key attribute. These cleanliness estimations can be stored and updated in the linkage graph. We now propose a mechanism to estimate the cleanliness of a join path using this error information.

We first define the cleanliness of an edge in the join graph. We first consider the join on textual values. Consider two linked nodes $R[X]$ and $S[Y]$ in the graph, where we align the values in $R[X]$ and $S[Y]$ using the maximum matching defined in Section 2.3. Suppose that the maximum lexical distances in the two nodes are respectively DIS $(R[X])$ and DIS $(S[Y])$. Hence, the distance between two values in $R[X]$ and $S[Y]$ that represent the same object should be within $\text{DIS}(R[X]) + \text{DIS}(S[Y])$. Thus we limit the distance between the pairs of aligned values in the maximum matching to $\text{DIS}(R[X]) + \text{DIS}(S[Y])$. The fraction of the values in the foreign key that appear in the maximal matching is an estimate for the cleanliness of the edge. Similarly, we can do the same thing for numeric values with a numeric metric. Given a join path, its cleanliness is obtained by multiplying the cleanliness values of all the edges and the nodes involved in the predicates

An extension of the above majority voting methodology is to construct a "disorder metric" that indicates the distance between an incorrect value and its ground truth. For numeric values, we use the average and standard deviation as the disorder metric. For example,

if salary errors average 5% with standard deviation 2%, then most of the errors are in the range from 3% to 7%. For text values, we use the "lexical distance" (such as distance based on WordNet [4, 17] and edit distance) as the disorder metric. For example, if an address field routinely confuses "*road*" and "*street*", but very rarely gets the name of the street wrong, then the lexical distance is very small. Using this tactic, we can use fuzzy predicates and fuzzy joins in place of the exact operations to improve the accuracy of our results. We plan to examine this tactic in detail as future work.

4.3 Query Planning with Data Cleaning

In a query plan with cleaning operations, the cleaning cost and the result quality are different based on the position of the data cleaning operation. For example, cleaning a whole column before the selection on this column yields a higher cleaning cost and query accuracy than putting the cleaning operation after the selection operation. Obviously expensive cleaning should be performed on as few records as possible and have as max impact as possible. We now give a way to obtain a query plan with cleaning operations that achieve high quality gain with a limited cleaning budget. The intuition is that for each value in each node of the join path, we count how many times it appears in the query result. Then we clean the values in the decreasing order of the number of times they appear, until our budget is exhausted. This will only address false positives; a different approach is required for false negatives. We leave it as a future work.

5. UPDATES

Real-world data is rarely static, and so some way to deal new datasets and updates is needed. We consider three types of updates.

(1) Insertions/deletions on source tables. This happens when there is a change to a table, e.g., the insertion of a new procurement record in the MIT Data Warehouse. This may also happen when data sources are cleaned (see Section 4).

(2) Replacement of source tables. Large companies typically rely on both internal and external information to build their knowledge bases. For instance, Merck collects published standard medical names from the World Health Organization (WHO) to help construct their own ontology. These data sources are updated periodically by WHO. Sometimes, even the format may be changed, e.g., from a JSON file to a CSV file.

(3) Updating MVs. MVs might be created based on other MVs. Since new data can arrive at any time, and cleaning can be done at any time to any record, MVs may need to be updated.

DATA CIVILIZER uses three corresponding strategies to cater to the above updates.

(i) MV maintenance. DATA CIVILIZER will need to incrementally propagate updates through the data curation pipeline to update downstream MVs, as well as from intermediate results that are cleaned back to data sources, if possible. To perform this propagation, we plan to leverage the techniques in DBRx [9], a system developed by QCRI and Waterloo. In some cases, the human effort to update and clean MVs may be daunting, and automatic methods may fail. In such cases, MVs should be discarded rather than updated.

(ii) Provenance management. To perform update propagation, DATA CIVILIZER will need to keep track of the relationships between data sets (their provenance). DATA CIVILIZER will leverage Decibel [16], a system developed at MIT for this purpose.

(iii) Incremental graph updates. As data changes, the linkage graph and profiles will also need to be updated. To support incremental updates to profiles we use a simple reference counting scheme, where each keyword has a count associated with it that is decremented when a word is removed. A word is removed from the profile when its count reaches zero. To support updates to the graph, rather than recomputing relationships every time an table changes, we maintain an estimate of the fraction of rows in a column that have changed since we last updated its node and edges in the graph, and re-index a column when this estimate goes above a set threshold.

6. TRACTABLE CURATION WORKFLOW

The process of data curation entails multiple iterations of discovery, linking, querying, and curation. Cleaning and transformation procedures must be guided by the user. DATA CIVILIZER will use a fairly conventional workflow system that will allow a human to construct sequences of operations, undo ones that are unproductive, and utilize branching to try multiple processing operations.

In addition, we plan to build a workflow orchestrator which will retain previous operation sequences and propose those that best fit a new situation. We expect this tactic to be successful because the types of data curation and preparation steps in an enterprise often follow repetitive patterns. Specifically, our workflow orchestrator will store the user query, the sequence of operations in a central workflow registry together with the meta-data and signatures provided by the data discovery component.

The orchestrator will evaluate a new user query and the initial results of the data discovery component against the workflow registry. Similar queries and similar data profiles are likely to demand similar cleaning procedures. Previous workflows will be ranked based on the similarity of the user query and retrieved discovery results. Accordingly, the related curation and preparation procedures will be shown in ranked order to the user.

7. DATA CIVILIZER IN THE WILD

We have built an initial prototype of DATA CIVILIZER containing the data discovery module and the linkage graph builder as discussed in Section 3 and 2. Ultimately, we will integrate DATA CIVILIZER with `BigDawg`, our implementation of a polystore, to realize the end-to-end system described in this paper. We have been working closely with end users to adapt our prototype to relevant real world problems. In particular, we have deployed a preliminary prototype of discovery in two different organizations. The MIT Data warehouse (MIT DWH) is a group at MIT responsible for building and maintaining a data warehouse that integrates data from multiple source systems. Merck, a big pharmaceutical company, manages large volumes of data, which are handled by different storage systems. In the following, we first outline the current state of the system and the modules that are currently deployed. We then provide details on each organizations' requirements, and how we are using DATA CIVILIZER to help them.

7.1 Current Data Civilizer Prototype

The current prototype runs as a server and can access data that resides in a file system or one or more databases. Clients connect to DATA CIVILIZER through a RESTful API. The current deployment of DATA CIVILIZER includes both the graph builder with the error-robust inclusion dependency discovery from Section 2 and the discovery component described in Section 3. Thus a user is able to submit queries that consist of attribute names, attribute values, or tables, and receive different types of results based on the selected similarity function. The similarity function depends on the specific use case, which we will discuss based on our two use cases MIT and Merck. We plan to demo our DATA CIVILIZER prototype at the conference.

7.2 MIT Data Warehouse

One of the key tasks of the MIT DWH team is to assist its customers – generally MIT administrators – to answer any of their questions about MIT data. For example, staff usually want to create reports, for which they need access to various kinds of data. The warehouse contains around 1TB of data spread across approximately 3K tables.

In their current workflow, a DWH customer presents a question, which a DWH team members by manually searching for tables containing relevant data. Once they have determined the tables of interest, they create a view that is accessed by the customer to solve the question at hand. Below are some of the common use cases we have encountered:

Fill in virtual schema. When a customer arrives with a question such as: *I need to create a report with the* ***gender distribution of the faculty per department and year***, the data warehouse personnel can use DATA CIVILIZER to find all the tables that contain schema names similar to the attributes exposed by the query, e.g., gender, faculty name, department, and year.

Table redundancy. Multiple views are created for different customers. Many of them contain very similar data, as multiple customers are interested in similar items. To reduce the redundancy of data, DATA CIVILIZER helps detect complementary as well as repeated sources. This sheds light on the status of the warehouse and helps to maintain it tidy and minimal.

Our prototype is deployed in an Amazon EC2 instance managed by the MIT DWH team with access to their data. We have been working with them for 3 months, iterating over priorities and learning about the problems they are facing.

In the future, we aim to deploy the entire DATA CIVILIZER system to enable running queries directly over the hundreds of data sources, by using the polystore query processing and curation module and linkage graph builder to create the necessary views on demand.

7.3 Merck

Merck is a large pharmaceutical company that manages massive volumes of data spread across around 4K databases in addition to several data lakes. One of the data assets of any pharmaceutical company are internal databases of chemical compounds and structures. Usually, these are more valuable when integrated with external, well-known and curated databases, such as PubChem [3], ChEMBL [1], or DrugBank [2]. We describe two common use cases that occur in this context:

Enrich data. One of the reasons for the existence of multiple chemical databases is that each puts an emphasis on different information. Analysts typically face situations in which they are interested in a set of attributes that are spread across different tables on different databases. DATA CIVILIZER helps to detect such attributes and bring them together *on-demand* to serve the users' purpose.

Identify entities. One single chemical entity may be referred to with different identifiers and formats in different databases. Chemical identifiers have been a subject of research in the bioinformatics community: multiple different formats have been proposed with different properties according to the scenario. DATA CIVILIZER helps them on this task by discovering datasets that contain schemas with the entities of interest.

We have been collaborating with 4 engineers and bioinformatics experts at Merck during the last 2 months. During this period we have learned about their use cases and we have used discovery on chemical databases that are publicly available. In the future, we plan to integrate internal datasets too, with the goal of evaluating DATA CIVILIZER with more varied datasets.

8. CONCLUSIONS

In this paper, we presented DATA CIVILIZER, an end-to-end big data management system. DATA CIVILIZER aims to support the discovery of data that is relevant to specific user tasks, the linkage of the relevant data for allowing complex polystore queries, the cleaning of the data under limited budget, the handling of data updates, and the iterative processing of the data. As data in large companies are usually scattered across multiple data storage platforms, we proposed the use of a polystore architecture to deploy our system. A key characteristic of DATA CIVILIZER is that it places the data cleaning operations in the query plan while trading-off the query result quality with the cleaning costs. We have deployed our preliminary system at two different institutions, MIT and Merck and obtained positive feedback from the users.

9. ACKNOWLEDGMENTS

Michael Stonebraker contributed significantly to the work presented in this paper; he is precluded from being an author because of CIDR rules concerning PC chairs.

10. REFERENCES

[1] Chembl. https://www.ebi.ac.uk/chembl/.
[2] Drugbank. http://www.drugbank.ca/.
[3] Pubchem. https://pubchem.ncbi.nlm.nih.gov/.
[4] Wordnet. https://wordnet.princeton.edu/.
[5] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.
[6] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. 24(4):557–581, 2015.
[7] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *ICDE*, pages 1134–1145, 2016.
[8] M. J. Cafarella, A. Y. Halevy, and N. Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.
[9] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, pages 445–456, 2014.
[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.
[11] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.
[12] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Çetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. G. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A demonstration of the bigdawg polystore system. *PVLDB*, 8(12):1908–1911, 2015.
[13] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *SIGMOD*, pages 795–806, 2016.
[14] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, pages 233–244, 1995.
[15] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011.
[16] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.
[17] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
[18] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.
[19] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 8(11):1310–1321, 2015.
[20] 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.
[21] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.