Building Data Civilizer Pipelines with an Advanced Workflow Engine

Essam Mansour[◇] Dong Deng[*] Raul Castro Fernandez[*] Abdulhakim A. Qahtan[◇] Wenbo Tao[*] Ziawasch Abedjan[†]
Ahmed Elmagarmid[◇] Ihab F. Ilyas[‡] Samuel Madden[*] Mourad Ouzzani[◇] Michael Stonebraker[*] Nan Tang[◇]

[◇]*Qatar Computing Research Institute, HBKU* [*]*MIT CSAIL* [†]*TU Berlin* [‡]*University of Waterloo*
{*emansour, aqahtan, aelmagarmid, mouzzani, ntang*}*@hbku.edu.qa*
{*dongdeng, raulcf, wenbo, madden, stonebraker*}*@csail.mit.edu* *abedjan@tu-berlin.de* *ilyas@uwaterloo.ca*

***Abstract*—In order for an enterprise to gain insight into its internal business and the changing outside environment, it is essential to provide the relevant data for in-depth analysis. Enterprise data is usually scattered across departments and geographic regions, and is often inconsistent. Data scientists spend the majority of their time finding, preparing, integrating, and cleaning relevant data sets. Data Civilizer is an end-to-end data preparation system. In this paper, we present the complete system, focusing on our new workflow engine, a superior system for entity matching and consolidation, and new cleaning tools. Our workflow engine allows data scientists to author, execute and retrofit data preparation pipelines of different data discovery and cleaning services. Our end-to-end demo scenario is based on data from the MIT data warehouse and e-commerce data sets.**

I. Introduction

Data analysts face several hurdles when applying their analytics tools on their data. Challenges include discovering data of interest, finding ways to combine the found data, detecting errors (including duplicates), and fixing these errors. Once they have a curated collection, they can proceed with their analytics. Discovery, integration, and cleaning clearly take a huge amount of time, up to 98% according to some of our collaborators [1]. We have been working for more than two years on building an end-to-end-system, Data Civilizer, to address these issues. We propose to demonstrate the complete system showing its different components and their interactions for a given integration task, focusing on our new workflow engine that facilitates the orchestration of various consolidation and cleaning tools and on our novel golden record component.

Data Civilizer contains three primary layers, namely the Civilizer frontend, the Civilizer engine, and the Civilizer services or modules. Figure 1 shows the architecture. The workflow engine allows a user to string together any of the Civilizer services in a directed graph to accomplish her data integration goal. Then the engine manages the execution of modules in the directed graph. Users interact with the system via a new GUI we have developed, called Civilizer Studio. In a nutshell, Data Civilizer has the following main services:

- A data discovery module, Aurum, to help find datasets of interest. Aurum builds an enterprise knowledge graph (EKG) to connect similar tables together. Aurum allows a user to browse the graph, run similarity searches on tables in the graph, and perform keyword searches. Aurum can also help find all join paths that connect these tables.
- An enrichment module, which will first union the results of all the join paths. In fact, we create the outer union of all the views, so we can keep all possible attribute values, thereby generating a sparse table. This module will then fill in some of the sparseness by joining the result with other available tables.
- An entity resolution module to form clusters of records thought to represent the same entity. We are currently using DeepER [2], a deep learning-based entity resolution module. This can be easily replaced with another module which can do the same job.
- A golden record module [3]. This component has the effect of collapsing clusters representing the same entity into single representative records. Our golden record system excels at cleaning when there are multiple sources of the same information.
- Other cleaning tools in the system include an abbreviation system [4], a disguised missing values detection tool [5], and ImputeDB [6] for filling in missing values.

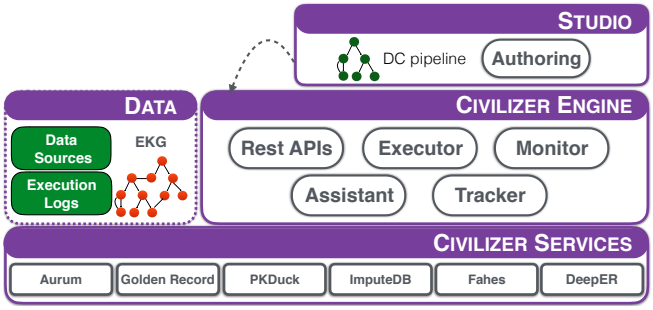

Figure 1. An overview of the Data Civilizer system

To better grasp the importance of Data Civilizer, we first discuss, in Section II, one of the scenarios that we will show during the demo. Section III outlines the main features of the workflow engine. We then highlight some details of Data Civilizer services in Section IV.

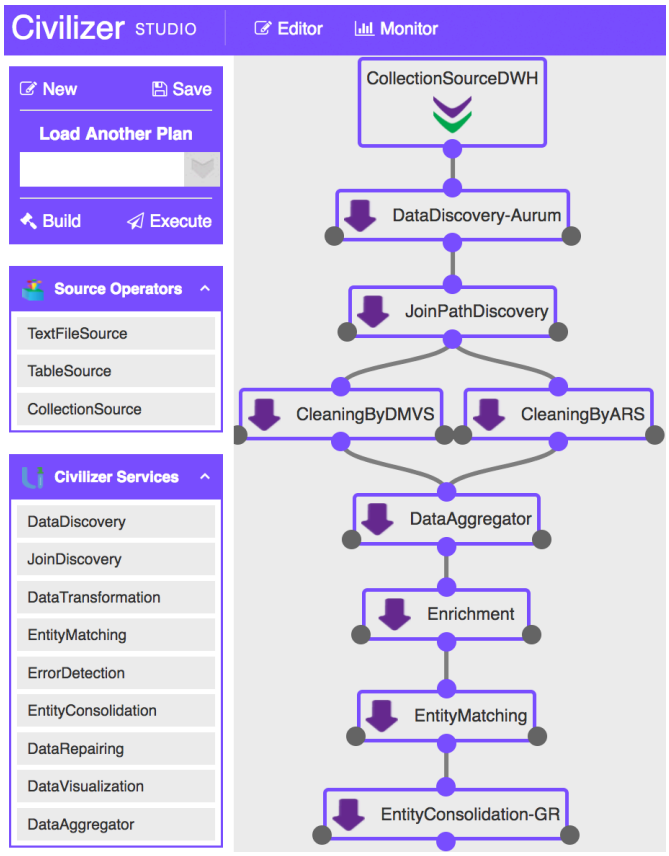

Figure 2. A sample DATA CIVILIZER workflow

II. PROPOSED DEMO SCENARIO

We describe our demo using the MIT Data Warehouse (DWH), which consists of 2,400 tables. The DWH contains typical university information, such as buildings, courses, faculties, students, departments, and schools. There are multiple independent organizations contributing to DWH which leads to duplicates, inconsistencies, and semantic violations. In our demo, we use a subset of 167 publicly available tables.

Consider the following use case. The Stata Center is a building on the MIT campus, which is mainly occupied by the Computer Science and Artificial Intelligence Lab, as well as a few other organizations, such as the Linguistics Department. A facility administrator may want to know the rooms in the Stata Center that are occupied by the Linguistics Department and a list of people assigned to each room. To accomplish this task, she can use DATA CIVILIZER with the workflow shown in Figure 2, which is discussed below.

1) DATA CIVILIZER allows data scientists to work across multiple data sources. The DATA CIVILIZER *source operator* creates a data access configuration file and identifies the corresponding EKG, or invokes the Aurum indexing service, if there is no EKG for this data. The *source operator* is for a data file, database table, or a collection of data sources. The *CollectionSource* allows a user to create a configuration file to fetch data from different sources, i.e., databases. Therefore, she starts the workflow by creating the *CollectionSourceDWH* node to connect to and index, i.e., build an EKG, DWH.
2) Then she can execute Aurum and specify queries to find tables containing room and department information. There are two such tables in the MIT DWH. *FacilityRoom* contains the space information and *SisDepartment* contains a list of MIT departments.
3) Now she can call the join path discovery service to find the tables that connect *FacilityRoom* and *SisDepartment*. In our demo, we find four connecting tables *MITStudentDirectory*, *EmployeeDirectory*, *SePerson*, and *WarehouseUsers* that link *FacilityRoom* and *SisDepartment*. The schemas of these "middle" tables are shown in Table I. This results in different possible join paths.
4) Next, she can send all the tables found in the join path to our data cleaning services, which generate updated tables. We provide different cleaning tools that can be used in parallel. The most useful cleaning tool here is our abbreviation resolution system (PKDuck), which can normalize entity names such as 'CS', 'CompSci', and 'Computer Science'. Our disguised missing values system (Fahes) can also clean disguised missing values. For example, the user may use DMVs with *WarehouseUsers* and *MITStudentDirectory* tables to clean columns, such as *OfficePhone* and *UnitName*. Notice that these cleaning steps may be done in parallel with separate human checkers. This step ends by aggregating the updated tables in a single data collection.
5) After that, the data enrichment service will union the four join paths. The user needs to specify the columns she is interested in with any filtering condition. For example, she might want to restrict the output to the Linguistics Department and rooms in the Stata Center. This will produce a single table containing all the needed information.
6) Th enrichment process is followed by entity matching as each person may appear in multiple tables. For example, an employee in the *EmployeeDirectory* table may also be a data warehouse user in the *WarehouseUsers* table. To remove duplicate information, this single table must be sent to the entity matching service, which will form possibly duplicate records into clusters.
7) Finally, the table of clusters is sent to our entity consolidation service to produce for each cluster, a single golden record corresponding to one person.

III. THE DATA CIVILIZER WORKFLOW ENGINE

Our workflow engine manages and monitors the data integration and preparation pipeline, and allows data scientists to work across multiple data sources. The workflow engine simplifies pipeline authoring by providing APIs at

EmployeeDirectory:	MITID	FullName	OfficeLocation	OfficePhone	DirectoryTitle	DepartmentNumber	DepartmentName	KrbName	EmailAddress	
WarehouseUsers:	MITID	KrbName	Name	EmailAddress	OfficeLocation	OfficePhone	UnitId	UnitName	Title	Type
MITStudentDirectory:	FullName	OfficeLocation	OfficePhone	EmailAddress	Department	DepartmentName	StudentYear			
SePerson:	MITID	KrbName	FullName	PayrollRank	PositionTitle	IsActive	OfficeLocation	Organization	EmployeeType	

Table I
INTERMEDIARY TABLES USED IN THE DISCOVERED JOIN PATHS

the service level and uses a table-in-table-out interface with the EKG as a global data structure for all DATA CIVILIZER services.

The workflow supports pipeline authoring using a GUI, called Civilizer Studio. The workflow functionality is exposed via RESTful APIs. Data integration tasks are organized into projects. Projects can be created and dropped at any time. Every workflow project is a directed acyclic graph (DAG) of service invocations. Each directed edge from node *u* to node *v* indicates that the node *v* will take the output of *u* as its input. The user can instantiate a node using different modules. For example, join path discovery could be instantiated from the Aurum service or the *ML4Join* service. This gives flexibility to the user to dynamically utilize different services. Our workflow can run parts of a pipeline in parallel to accelerate the process and make the most of the available resources.

DATA CIVILIZER services need human-in-the-loop validation. Naturally, these may take a long time to finish. DATA CIVILIZER tracks and logs at two levels, (i) at the module level, and (ii) at the workflow level for the overall interaction. Any new module has to implement the DATA CIVILIZER tracking and logging protocol. At the module level, the workflow engine maintains a tracker, which logs information for all the fine-grained operators used inside the module. This allows nodes to be restarted after failures and allows the effects of a module to be undone if desired.

The *Civilizer Assistant* can guide a user in constructing a workflow that suits a certain analytical task. Specifically, we propose to remember all past workflows, all past source data, all past curated results, and any intermediate results in a workflow pipeline that might be useful. This Curation Database (CD) can be accessed by extensions of all of our services to turn them into "learn from history" services. For example, our golden record service processes each cluster of records and finds "meta rules". When applied, these rules can alter values to reduce or eliminate "non golden" values. Remembering all rules that have been previously verified will allow an ML system to automatically accept many of these rules, resulting in a system with higher accuracy and less human intervention.

IV. THE DATA CIVILIZER SERVICES

We briefly describe the different services of DATA CIVILIZER which will be demonstrated. For some of these services, more details can be found in their respective publications.

A. Aurum

The MIT DWH administrators react to requests for information from across the Institute. In effect, these are ad-hoc queries to a 2400 table database. The DWH administrators admit they spend considerably more time finding the information relevant to a user's request than in writing the SQL that responds to the request. Hence, they have a "data discovery challenge". Other companies we are working with including Merck, British Telecom, and the City of New York report the same discovery challenge.

We designed Aurum to build, maintain and query an Enterprise Knowledge Graph (EKG). This graph contains a node for each table and each column and edges that maintain relationships between nodes. Building the EKG requires accessing data sources repeatedly. To minimize such accesses, we introduce a two-stage process consisting of a profiler that summarizes all data from sources into space-efficient signatures, and a graph builder, which finds syntactic relationships, such as different similarities, and PK/FK candidates in $O(n)$ using only the signatures.

Aurum can discover join paths based on the constructed EGK. We developed another method for join path discovery, called *ML4Join*, based on machine learning techniques. *ML4Join* extends the method proposed in [7] to detect approximate PKFK edges. That allows ML4Join to avoid the data heterogeneity.

Finally, Aurum contains a source retrieval query language (SRQL) based on a set of discovery primitives that can be composed arbitrarily, allowing users to express complex discovery queries and allowing them to express different ranking criteria for results. Links in the EKG are also enriched using ontologies thus allowing the linking of tables and columns that would otherwise not be connected. This further helps users find datasets of interest.

B. Enrichment

Given two tables, Aurum can produce all the join paths between them using information in the EKG. Then, the data enrichment module materializes these join paths and unions the results into a big, wide and presumably sparse table. Because the materialized tables contain different attributes, some parts of the union table are NULL. To fill in missing values, enrichment uses the join path discovery service to look for additional tables in the EKG that can join with the result table using attributes in the result. From the set of all possible join paths, enrichment chooses those that fill in some of the null values without producing different values

for already populated cells. In the future, we plan to look for more complex mechanisms to choose enrichment tables.

C. Golden Record Selection

Using an entity resolution tool, the records in the union table are partitioned into clusters that represent the same entity. The next step is to reduce each cluster to a "golden record", which contains the canonical values for each attribute, a.k.a the entity consolidation problem. Truth discovery systems are often used to solve this problem, usually employing heuristics such as majority consensus (MC) or source authority to determine the exemplar record. However, such techniques are not capable of resolving simple data variation, such as *Jeff* ↔ *Jeffery*, and may give biased results.

To address the above issue, we implemented algorithms to reduce clusters using automatically generated matchings. These are substring pairs that can replace each other, e.g., *9th* ↔ *9* and *Jeff* ↔ *Jeffery*. To this end, we first align the value pairs within a cluster by their longest common subsequences to generate candidate matchings. Then we aggregate candidate matchings with common syntactic characteristics (such as *9th* ↔ *9* and *15* ↔ *15th*) into groups, including their syntactic structure and the program which describes how one side of the matching is transformed to the other. Finally, we solicit a human to validate these matching groups and apply the approved ones to transform values. The net effect is to lower the variability inside each cluster, allowing MC to do a better job producing golden records. Note that MC could be replaced by a more sophisticated truth discovery scheme if desired. Refer to [3] for more details on the algorithm and for experiments that show our enhanced MC scheme outperforms vanilla MC on a variety of data sets by up to 80%.

D. Other Cleaning Tools

We developed three specialized data cleaning tools, namely, PKDUCK [4] to deal with abbreviations, Fahes to detect and fix disguised missing values, and ImputeDB [6] to deal with missing values. These three data cleaning tools can be invoked at any stage of the workflow to improve data quality.

PKDUCK [4] is a join operator capable of finding similar string pairs between two columns that contain a mix of abbreviations and full values. For example, suppose a column contains dept of computer science and another column contains department of cs. PKDUCK can identify that the second value is an abbreviation of the first one and replaces such abbreviations with full values. Our module uses a novel similarity measure to quantify the similarity between two strings, given an abbreviation dictionary learned from input strings. This replacement is made as long as the similarity is greater than a threshold. An efficient join algorithm using the signature framework [8] makes PKDUCK scale to very large datasets. PKDUCK can be applied as part of the transformation function to help normalize values, or in the data discovery module to identify links in the EKG.

Fahes [5] looks for erroneous values that replace the missing values, which are known as disguised missing values (DMVs). Fahes accurately detects the DMVs using the following approach. We first use a data profiler to collect statistics about the data. These statistics are consumed by a detection engine which includes three main modules. Syntactical outliers and repeated pattern discovery module to detect data values with patterns that do not fit the data. For example, a phone number attribute might contain 1111111111, 1212121212, 1234512345 or ?. An outlier detection method is utilized to detect the DMVs that are out of the range. For detecting DMVs that follow a missing-at-random (MAR) model, an improved version of an existing algorithm [9] is used.

ImputeDB [6] performs on-the-fly imputation of missing values while processing a query. Traditional mechanisms either impute the whole database which is expensive, or drop all tuples containing missing values, which can possibly introduce bias into the data and may discard an excessive number of tuples. ImputeDB, on the other hand, frees analysts from a base-table imputation step by adding two types of operators into the query plan: Impute and Drop. The Impute operation fills in missing values using any statistical imputation technique. The Drop operation simply drops tuples which contain missing values. A cost model is used to measure the quality of the imputation operations. A query planning algorithm is also designed to jointly optimize for running time and result quality. See [6] for results on the performance of ImputeDB.

REFERENCES

[1] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang, "The data civilizer system," in *CIDR*, 2017.

[2] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang, "DeepER - Deep Entity Resolution," *CoRR*, vol. abs/1710.00597, 2017. [Online]. Available: http://arxiv.org/abs/1710.00597

[3] D. Deng, W. Tao, Z. Abedjan, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, "Entity Consolidation: The Golden Record Problem," *ArXiv e-prints*, Oct. 2017.

[4] W. Tao, D. Deng, and M. Stonebraker, "Approximate string joins with abbreviations," *PVLDB*, vol. 11, no. 1, 2018.

[5] A. A. Qahtan, A. Elmagarmid, M. Ouzzani, and N. Tang, "FAHES: Detecting disguised missing values," in *ICDE*, 2018.

[6] J. Cambronero, J. K. Feser, M. J. Smith, and S. Madden, "Query optimization for dynamic imputation," *PVLDB*, vol. 10, no. 11, pp. 1310–1321, 2017.

[7] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser, "A machine learning approach to foreign key discovery," in *WebDB*, 2009.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006, p. 5.

[9] M. Hua and J. Pei, "Cleaning disguised missing values: A heuristic approach," in *KDD*, 2007.