The Data Civilizer System

1st Author

1st author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

1st author's E-mail address

2nd Author

2nd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

2nd E-mail

3rd Author

3rd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

3rd E-mail

**ABSTRACT**

**CCS Concepts**

• **Information systems➝Database management system engines**   • **Computing methodologies➝Massively parallel and high-performance simulations.**This is just an example, please use the correct category and subject descriptors for your submission*.* The ACM Computing Classification Scheme:

<http://www.acm.org/about/class/class/2012>. Please read the [HOW TO CLASSIFY WORKS USING ACM'S COMPUTING CLASSIFICATION SYSTEM](http://www.acm.org/publications/article-templates/CCS-HOWTO-v6-12Jan2015.docx) for instructions on how to classify your document using the 2012 ACM Computing Classification System and insert the index terms into your Microsoft Word source file.

**Keywords**

Keywords are your own designated keywords separated by semicolons (“;”).

# INTRODUCTION

An often cited statistic is that data scientists spend at least X% of their time finding, preparing, integrating and cleaning data sets. The remaining 100-X% of their time is spent doing the actual desired analysis tasks. X is reputed to be 80, 90, 95, or even higher; in fact, a data officer (Mark Schreiber of Merck) estimates X to 98%, in which case data scientists spend less than one hour a week on tasks in their job description.

To address various aspects of this problem, several systems have been developed. For example, Data Wrangler [10] and DataXFormer [2] automate data preparation by enabled systematic data sources; Data Tamer [13] attempts to integrate and unify large disparate data sources via schema mappings and record linkage exercises to help scientists work with data sets across silos; DeepDive [12] extract facts and structured information from large corpora of text, images and other unstructured sources. However, these prior systems all assume that the data scientist already has a small number of tables he or she wants to clean, integrate, or extract data from, leaving a key challenge unaddressed: finding the data of interest. Since a typical enterprise data lake contains thousands of heterogeneous and ill-specified tables, even answering simple questions about where to start looking is hard to do.

SAMPLE: Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference’10*, Month 1–2, 2010, City, State, Country.

Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

DOI: http://dx.doi.org/10.1145/12345.67890

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 X, by helping data scientists quickly find and query data of interest from large numbers of tables, allowing data scientists to spend more of their time on productive data analysis. We present the main design requirements of DATA CIVILIZER and the challenges it entails through use cases from the environment at Merck, a large multi-site drug company with about XXX employees, of which YYY are data scientists.

**[Discovery]** A data scientist at Merck has a hypothesis, for ex- ample, *the drug Ritalin causes brain cancer in rats weighing more than 1 kg*. 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. A Discovery component in DATA CIVILIZER (cf. Section 3) has to assist the scientist with finding data sets of interest.

**[Stitching]** The linkage between different data can be extremely helpful for the users to find the interesting data. Data stitching is in charge of finding these linkages. Given *n* datasets, in worst case there are linkages among them. Thus data stitching runs offline and utilizes data discovery to prune the search space in order to run at a small scale. The linkage can be any significant relationship between two datasets, such as data equivalent and data subsuming. Among them, the most important one is the functional dependency, i.e., the primary key and foreign key relationship. We discuss data stitching in detail in Section 4. When the users identified his interesting data, data stitching can also generate a view across these data for the user. It is now necessary to perform data curation on these views by, for example, extracting data from source data storage systems, performing schema integration on the multiple views, transforming data into a common representation, cleaning erroneous values from the source data sets, and performing entity consolidation on resulting records. Multiple tools, including the tools we mentioned earlier, can be used in tackling these curation and extraction challenges. We have recently evaluated some of the available cleaning tools on multiple data sets [?] to identify the main challenges in this domain.

**[Curation Polystore]** Since Merck has a variety of massive-scale data storage systems, it is not feasible to move all data to a central data warehouse. Also, it is not economically, nor technically practical, to perform data curation of all of the thousands of databases up front. Instead stitching and data curation in DATA CIVILIZER must be done on-demand, as data scientists need to access data to get their work done. Therefore, we are building DATA CIVILIZER using a polystore architecture [7], which can pull data out of multiple underlying storage engines as needed and as dictated by the current analytics task. Obviously, data cleaning, data transformation and entity consolidation must be integrated with querying the polystore. In this way, a key technical optimization is to push filters and joins through cleaning operations and into the underlying data storage sys-tems wherever possible. The merger of polystores and data curation steps is discussed in Section 5.

**[Cleanliness Estimation]** Because of the human effort involved, the expensive processes of creating and curating materialized views can be very disruptive to the human interaction experience (e.g., validating the view definitions and the stitching results, and validating the automatic cleaning decisions and suggestions). Therefore, DATA CIVILIZER must estimate the cost of constructing and curating such materialized views to reason about the feasibility of these operations within the scientists’ time budget. Estimating the cleanliness of a view entails constructing a model for how dirty the data in the source data sets actually is, which we discuss in Section ??. A related topic is estimating the cleanliness level that can be achieved for a given cleaning budget. In this way, a scientist can decide whether or not he wishes to proceed with the project at hand.

**[Optimizing Stitching]** It is highly inefficient and wasteful to discard expensive-to-construct materialized views after their initial use by a data scientist. Hence, we assume that they are generally retained for future use. Moreover, future materialized views may be based off previously constructed ones or on original data sources. As a result, there may be several ways to construct a new view, with different costs and accuracy. Therefore, the data stitching problem must be revisited to deal with this materialization cost/accuracy trade-off. This is the subject of Sections ??.2

**[Handling Updates]** If a source data set is updated, then 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. Lastly, if a scientist updates a view, we must propagate changes to other derived views, as well as back upstream to data sources, if this is possible. Section 6 discusses the update issues and how DATA CIVILIZER handles it.

**[Workflow]** To have end-to-end functionality, DATA CIVILIZER has to offer a workflow engine whereby data scientists can iterate over our components in whatever order they wish; they should also be able to undo previous workflow steps and perform alternate branching from given materialized views. Section 7 discusses the workflow management in DATA CIVILIZER.

We describe the current implementation of DATA CIVILIZER in Section 8, and indicate initial user experience in two use cases: the MIT data warehouse and Merck. We conclude with final remarks and an outline of our future research plans in Section 9.

# ARCHITECTURE

DATA CIVILIZER adopts the main design principles mentioned in the introduction. It is built as a stack of modules: Discovery, Stitching, and finally a Query Processing and a Workflow Management layer. All layers assume a polystore computing environment. Figure 1 shows the general architecture of the DATA CIVILIZER system. The discovery layer provides search facilities over the underlying slew of available data sources. It manages the scale of the underlying data using a high-performance profiler that operates on data at different granularities using summarization and statistical profiling techniques. Once profiles are build, the discovery engine find related groups of relations using measures such as schema or content similarity; this is responsibility of the graph builder component. These relationships are represented in a data fabric similar in spirit to a knowledge graph [3, 4, 14].

The output graph is later used by the stitching layer to find more complex relationships to stitch these units together and answer a wide range of analytic queries against ad-hoc schemas. The stitching layer focuses primarily on exploring the large space of possible “join relationships” that connect the underlying data sets. For example, discovering (often fuzzy) primary key-foreign key relationships between tables with a cleanliness estimator. More generally it extracts inclusion dependencies through the FK-PK finder, which allow for joining or stitching these raw data sources together to populate an analytics schema or a view3.

The query answering and workflow management layer supports the answering of users’ queries and composition of user-defined workflows. Queries must be answered despite the heterogeneity and the different levels of trustworthiness and cleanliness of the underlying sources. This is possible because the previous layers provide the capability of creating schema views on-demand for the query to process by using the graph structure populated by the discovery and stitching layer. To create the right view, a module called workflow orchestrator is responsible for coordinating discovery and stitching. After executing a query (query processor), the results will be accompanied of certain quality, which can be improved by, for example, performing some data cleaning on key tables that contribute to the results. The module responsible for assessing the result quality and recommending improvement strategies is called quality controller.

The layers in DATA CIVILIZER have different scale and response time constraints, which greatly affect the design and the choice of algorithms in each layer. The discovery layer is “always on” working in the background to find, index and mine connections among large number of data sets, hence, almost-linear algorithms that work on a Big data scale is a key requirement. The stitching layer has to find more complex relationships, such as inclusion dependencies, which are too expensive to run on the full scale raw data. Hence, the stitching layer has to judiciously use the graph output of the discovery layer to scope the search space. While the stitching layer has an off-line component that focuses on mining these complex join relationships, efficient use of the query submitted o the query processing layer can help pruning the large space of possible join graphs. Finally, the query processing layer has the tightest response time constraints to respond within reasonable time to ad-hoc analytics against often continuously changing analytics schemas (views). Limiting the space of possible data stitching strategies, while taking into account the polystore execution environment is a key technical challenge.

# DISCOVERY

The goal of the data discovery module is to find relevant data among the millions of datasets spread across many storage systems of modern organizations. Suppose an analyst wants to answer the question: what is the monthly sales trend by department? The analyst knows conceptually what data is needed to answer this question (a table of sales, a table of departments, a table of products sold by each department), but not which specific data sources (which relations in a schema or what files in an HDFS deployment) contain such data. The typical solution is to 1) ask an expert (if such a person is available), or to perform manual exploration, inspecting datasets one by one (which is time-consuming and prone to missing relevant data sources). We say this analyst is facing a data discovery challenge.

The data discovery module narrows down the search of relevant

data from the thousands and millions of data sources to a handful of them that can then be fed to stitching—that performs the final preparation before processing. Discovery exports an API that can be used by users directly to search for relevant data, e.g., schema search, similar content, etc, as well as by other modules in Data Civilizer, such as stitching, to retrieve additional data that can be used, for example, to enrich a table. Next we describe the general steps of data discovery:

## Data Discovery Components

The data discovery module consists of two conceptually different components that collaborate with each other to build a skeleton of the linkage graph shown in Figure 1. The linkage graph is defined as G = (V,E), where E is a multiset, as we permit different types of edges, i.e. relationships, betweeen V . To build this multigraph, discovery consists of two modules, the first one, the data profiler is in charge of finding the V in the graph, and the second one, the graph builder is responsible for finding E.

**1) Data profiler.** We first accumulate knowledge of the data sources by summarizing them into concise profiles. We can choose the granularity of those profilers. In practice, we have found that building a profile per attribute is sufficient for most use cases. A profile contains a signature, which is a domain- dependent, compact representation of the original content. One example of signature for numerical data is its distribution, and for textual data a vector with the most significate terms. A profile also contains information about the data cardinality, data type, and numerical ranges when it applies [1].

**2) Graph builder.** The data profiler generates a set of V , and the graph builder responsibility is to find relationships among these, and represent them as E, building the skeleton of the linkage graph. These relationships help to navigate through the different sources (V ). Examples of relationships that the graph builder finds are content similarity based on measuring the similarity distance among the signatures that represent each v. Another relationship is schema similarity that captures the similarity of the names of the different v. Other kind of relationships capture the hierarchical relationship among v, for example, all attributes of a same relation are connected, permitting quick navigation. Creating these relationships require in general a pairwise comparison that would render the module too slow. We discuss next some techniques we use to tame the scale.

Once the linkage graph skeleton has been built, it can be accessed by a set of data discovery primitives to explore and find relevant data. These primitives are used by users and by other modules, such as stitching. In particular, stitching also has write access to the linkage graph, that uses to add a new e to the multiset of relationships, the expensive-to-compute FK-PK relationships. Data discovery primitives can be composed into more complex data discovery functions, and all of these combine through combinator operators to build expressive discovery queries.

## Data Profiler

The profiling module of data discovery is responsible for com- puting signatures for each attribute in the dataset, as well as data types, cardinalities and an estimation of the cleanliness of the data, i.e. a measure of its data quality. Signatures are type-dependent and must exhibit two properties: (i) represent the attribute values in a compact way; and (ii) be compatible with a similarity metric.

Signatures can be of two types, numerical and textual. For nu- merical values the module learns the probability distribution of the data and it also computes the data range by estimating the median and inter-quantile range, i.e. the 75 p − 25 p. In the case of textual data, the module computes the TF-IDF vector for each profile of data, capturing the terms that better represent the attribute.

Fast cardinality and quantile estimation. Due to the sheer volumes of data, discovery relies heavily on sketches to estimate the cardinality of the attributes as well as the quantiles in the case of numerical data. This avoids ordering the data. More in general, this permits the module to compute the profiles following a read-once principle, saving memory and computation time. By treating the input data as a read-once stream we also avoid putting too much pressure on the original data sources.

Data quality estimation. Along with data cardinalities and quan- tiles, that can help to identify candidate outliers, the profiling module also collects information about common data errors, such as empty values. When safe, it performs some denoising, i.e. removing empty values, leading to better quality profiles, and then it includes this cleanliness information as part of the profile. This serves as a prelim- inary step to other more-expensive cleaning operations performed by other modules in data civilizer, such as stitching and the query processor ??).

## Graph Builder

The input of the graph builder is a set of profiles, V , that are the nodes of the multigraph represented by the linkage graph. Its repon- sibility is to compute the multiset E of relationships among each pair of v. In particular, the graph builder component finds content and schema similarity relationships. In this way, the multigraph will have an edge of type content-similarity between v A and B if these are similar, and one of type schema-similarity if the schema names are similar. Each edge is scored with a weight that represents the strength of the relationship: the particular meaning depends on the edge semantics.

This approach creates a logically complete graph, where all nodes in the graph are connected with at least some similarity among them. In practice, it is possible to define a minimum similarity threshold that must be surpassed for an edge to exist in the graph.

Finding the edges among V requires a pairwise comparison among all v in V, which is computationally very expensive (O(N2)), and due to the large scale of datasets that discovery must process, irrealizable. To tame the scale we rely on approximate clustering techniques such as locality sensitive hashing (LSH) [6]. With this approach we hash the signature that represents each v in V and we group together those that collide, i.e. that are more likely to be similar. Then we perform a comparison among the v in each of the clusters to finally determine whether a relationship exists or not. This approach avoids the N2 operation and still achieves high accuracy.

The entire discovery layer benefits from a distributed architecture that helps to parallelize the profiling as well as the graph building module, further speeding up the process. Other kind of relationships such as finding FK-PK are too expensive to compute by discovery, and are left to stitching, that follows a more judicious approach to choose which ones to compute, as explained next.

# DATA STITCHER

The linkages between different data can be extremely helpful for the users to find the interesting data. Data stitching links all the data together and adds these linkages into the graph. Among all the data linkages, the primary key and foreign key (PK-FK) relationship is one of the most important. Data stitching first utilizes the inclusion dependency to find the candidate PK-FK relationship and then refine the candidates by advanced techniques. However, as we all known, data in the wild is rather dirty. The dirty data contaminates the inclusion dependency in two ways: make fewer keys overlap and make keys not exactly match. To tolerate errors, we extend the traditional inclusion dependency by both key coverage and text similarity and propose the *error-robust inclusion dependency*.

Note it is time consuming to find the error-robust inclusion de- pendency. To address this issue, data stitching leverage the data profiling results from data discovery to reduce the search space. Moreover, as the advanced techniques that refine the PK-FK candidates are computation intensive, we only apply them after the users identified his interesting data.

Data stitching also estimates the cleanliness of the linkage in the graph, which can help curate the ploystore query.

## Error-Robust Inclusion Dependency

Foreign key is one of the most important schema information in managing and using data, which is typically missing in real world. The foreign key and primary key relations are usually identified by inclusion dependency. However, data in the wild is full of errors, such as inconsistence and different formats. This yields to the requirement of error tolerating in inclusion dependency. We observe that the errors in data can contaminate the inclusion dependency in two ways. First, they make the corresponding primary keys and foreign keys not match exactly. Second, they make the foreign keys not all covered by the primary keys. To address these issues, we design an error-robust inclusion dependency which enhances the traditional inclusion dependency with value coverage and text similarity. Next we give the details.

Given two projections R[X] and S[Y], we build a weighted bipar- tite based on the projections. There is a bijection between the vertexes in and the distinct instances in R[X], i.e., each vertex u in maps to a distinct instance in R[X ] and vice versa. This also applies to the vertex set and the distinct instances in S[Y]. For any vertexes and , there is a weighted edge where the weight is defined by the text similarity between their corresponding instance values, such as Jaccard similarity and edit similarity. Let be the *maximum weighted bipartite matching* of and

Then EIND(R[X ], S[Y ]) is proportional to the chance of an inclusion de- pendency from R[X ] to S[Y ]. Given an error-tolerating threshold , we formally define the error-robust inclusion dependency as follows.

**Definition 1** (Error-Robust Inclusion Dependency). *Given two projections R[X] and S[Y] on relational tables and an error-tolerating threshold δ, there is an error-robust inclusion dependency from X to Y if and only if EIND(R[X], S[Y]) ≥ δ.*

Note the value domain of EIND(R[X], S[Y]) is [0,1]. there is an exact inclusion dependency among the two projections if and only if . When the projections cross multiple fields, we can utilize different text similarity functions on different fields. As it requires all the fields to be match in inclusion dependency, we combine the text similarities in different fields by multiplying them. For example, given two instances (SIGMOD Conference, Sam Madden, San Francisco) and (SIGMOD Conference 2016, Samuel Madden, San Francisco). Suppose that we use Jaccard similarity to evaluate the first field and edit similarity to evaluate the second field. Then we can combine the text similarities as The score can also serve as the cleanliness estimation of the linkage between R[X] and S[Y].

## Refine Candidate FK-PK Relationships

The error-robust inclusion dependency gives us a bunch of can- didate foreign key and primary key (PK-FK) relationships. We can apply existing machine learning method to remove the false positives from the results.

# POLYSTORE QUERY PROCESSING

Our local polystore system is called BigDAWG. It consists of a middleware query optimizer and executor, and shims to various local storage systems, as noted in [7, 8]. Assume that a user has run discovery and stitching to identify a virtual relation (view) that he wishes to query. Also, assume that he has identified the subset of each source table in which he is interested. In other words, he has defined the join path using stitching and the predicates that subset the tables in some other way. This will translate directly to a BigDAWG query. In general, a user will want to retrieve a subset of the view, which is merely an extra predicate(s) attached to the BigDAWG query. In this section we address how to integrate data cleaning and transformation operations with this query plan.

**Example.** To achieve high quality results, one has to, in general, clean the data prior to 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 an automatic algorithm. Hence, it is expensive in human resources, which we believe is generally “the high pole in the tent”. As such, a user has to decide how he wants to trade off data quality and cleaning cost. Data Civilizer defines two parameters under his 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 the 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, he can use P and M to quantify the value of his time. In other cases, cleaning is done by other domain experts, who generally need to be paid. In this case, P and M are statements about budget priorities.

Obviously one want to perform expensive cleaning on as few records as possible. Hence, one would like to insert cleaning opera- tions as late in the query plan as possible. Unfortunately, if one runs a query that has the predicate:

. . . where name = “New York”

then the misspelled city name will not be found, and accuracy will suffer. One solution is to clean the entire source data sets to avoid such errors, an expensive proposition indeed. In Data Civilizer, we support a third alternative; namely more information about the errors in each data set. First, we assume that each data set owner gives us accuracy metrics for each column, namely an estimate for the percentage of the column which is erroneous. Second, the same data set owner is required to specify a “disorder metric”, which indicates the average and variance of the lexical distance between an incorrect value and its ground truth. For example, if salary errors average 5% with variance 2%, then 2/3 of the errors are less than 7% off. If an address field routinely confuses “road” and “street”, but very rarely gets the name of the street wrong, then the lexical distance is again very small. If the penalty is small, then we can insert cleaning “late” in the query plan. On the other hand, if the penalty is large, then we must insert cleaning earlier, of course at much higher cost. Next we introduce Data Civilizer’s cleanliness model and then we discuss how to achieve a target result quality.

## The Cleanliness Model

A user of Data Civilizer can specify attributes of interest that compose a query schema and predicates on the schema to select the data of interest. To answer a query, data civilizer generates a steiner tree for the query schema from the join graph. For a given query, there may be multiple steiner trees. Data Civilizer chooses the best one according to the following cleanliness model.

Assume R is the desired table, i.e. the ground truth, and R′ be the one generated from our join graph. Let |R| be the number records in R and |R ∩ R′ | be the number common records shared by R and R′. We respectively define the precision and the recall of the query result as

Precision = and Recall =

Next we give a cleanliness model to estimate the two values. Consider two tables A and B with FK-PK relationship in the Steiner tree. Let A.a be the foreign key and B.b be the primary key. Data stitching will give us a maximum bipartite matching between the values in A.a and B.b where the weight of an edge is the likelihood the two end points represent the same entity. We can join all the tables in the steiner tree by the maximum bipartite matching to achieve R and |R|. Next we estimate |R ∩ R′|. As each record in R comes from connecting all the nodes in the steiner tree, we multiply all the weights in the edges in the steiner tree as the likelihood that this record is in R ∩ R′. We aggregate all the likelihoods of all the records in R as an estimation of |R ∩ R′ |, such as summation. In this way, we can estimate the precision of the query result.

Next we estimate the recall of the query result, which can be achieved by estimating |R′|. To this end, we traverse the steiner tree in post-order. Suppose we are visiting a node n, whose parent is p. If n is the foreign key, we set the cardinality of p as the size |n| of table n....

## Maximize Quality Gain with a Budget

After the result for a query is generated, the user may want to improve the quality of the results, i.e. the quality of the steiner tree. Improving quality has a cost: we discuss in this section three alternatives to achieve such goal.

We first define a cleaning operation and its cost. We involve human to clean the data by asking them questions. We ask two kinds of questions. The first one is “Does tuple t satisfy a predicate P?” and the second one is “Are tuple t and t’ the same object?”. Assume all the questions have the same cost. Then given a budget, we can only ask the human a limited number of questions.

As we only give an estimation of the query result quality, after the user cleans some data, our estimated result quality can decrease in some cases. This is because our estimation on the more clean data is more accurate than our original estimation, which may have overestimated the quality. In general we do not want this situation to happen—it would be discouraging for the user to observe a real decrease of the query result quality after performing some cleaning effort. To solve this problem data civilizer offers three options:

1. To show estimation confidence along with the data quality. In this way, we can guarantee that at least the lower bound of result quality always improve after cleaning data.

2. Provide a minimum cleaning effort necessary to guarantee an estimated quality increase. The user can assess then whether it is feasible to carry on such effort.

3. Provide a set of specific data to be cleaned by users. Data civilizer guarantees in this case that the estimated quality will not decrease at any time when the users clean the data using our plan.

# UPDATES

Real-world data is rarely static, which is exactly the scenario that Data Civilizer faces in Merck and the M.I.T. Data Warehouse.

We categorize three types of updates managed by Data Civilizer.

**(1) Insertions/deletions on source tables.** This happens when there is a change of the table, e.g., insertions of new procurement records in the M.I.T. Data Warehouse. This may also happen when some data sources get cleaned or transformed on demand (see Sec- tion 5).

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

**(3) Updating MVs.** MVs might be created in cascade, and the human effort for data curation might happen in any layer.

In response to the above three types of updates, Data Civilizer uses three strategies correspondingly.

**(i) MV maintenance.** In the simplest case of small changes over the source data, such as the case (1) above, Data Civilizer will leverage the mature techniques for maintaining materialized views (see [9] for a survey), which has been widely deployed in many commercial DBMSs. In such a way, Data Civilizer incrementally propagates the updates through the data curation pipeline to update downstream MVs.

**(ii) Provenance management.** In some cases such as the above case (2), the human effort involved may be daunting for updat- ing the MVs. In these scenarios, the MVs should be discarded rather than updated. Naturally, there is need for a component that natively supports the versioning or branching of data to enable con- current analysis, cleaning, integration, or curation of data across data sources. Data Civilizer leverages Decibel [11], a system developed by MIT, for this purpose.

**(iii) Descriptive and prescriptive data cleaning.** Sometimes, a scientist may curate directly his MV such as the above case (3), which triggers some updates that must be propagated to other de- scendent data sets, as well as back upstream to data sources. To perform this, we leverage the technique in [5], a system developed by QCRI and Waterloo. In the MV, the updates will be captured based on human data curation, which will be transformed at the source level to prescribe actions to solve them.

As the data comes to us gradually while finding the inclusion dependency, especially the error-robust inclusion dependency, is time consuming, it is necessary to have the error-robust inclusion dependency incrementally founded. Moreover, the inclusion de- pendency does not necessary yields primary key and foreign key relationship, further techniques to eliminates the false positives are needed. After that, it is straightforward to construct the join graph using the primary keys and foreign keys: each table is a vertex and there is an edge between two vertexes only if there exists a primary key and foreign key relationship between them.

Table 1: Deployment environments of Data Civilizer[ra: I’d remove use cases from here]

# TRACTABLE CURATION WORKFLOW

The process of data curation on demand might entail multiple iterations of data discovery, data stitching, and data curation routines. Especially the data curation requirement might consist of various cleaning and transformation procedures that have to be guided by a user. To facilitate the user in this is process, the workflow orchestra- tor of Data Civilizer will propose operation sequences that best fit the current data set. To this end, we exploit the fact that the type of data curation and preparation steps inside a company might follow repetitive patterns.

The workflow orchestrator stores the user query, the sequence of component operations and data curation steps into central workflow registry together with the meta-data and signatures provided by the data discovery component. These information will be evaluated during future data discovery scenarios. The orchestrator will evaluate the current 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. The workflow orchestrator leverages the same similarity metrics that are used by the data discovery system. 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 a ranked overview to the user. As several workflows might be generated through different users for the same query the orchestrator will filter workflows that can be subsumed by existing simpler workflows. For this purpose, the workflow orchestrator tracks the profile change of discovered and stitched datasets after each operation to be able to identify the individual impact of each in transforming a data set. Additionally, it will take into consideration at which point the results of a component were accepted by the user.

# DATA CIVILIZER IN THE WILD

We have deployed a preliminary prototype of Data Civilizer in two different organizations. The MIT Datawarehouse is a group inside 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 managed by different storage systems. Some characteristics of both organizations’ data are shown in Table 1. Next we provide detail on each organizations’ requirements, and how we are using Data Civilizer to help them.

## MIT Data warehouse

The MIT Datawarehouse is a team within MIT that manages a warehouse that integrates data from hundreds of source databases from around the campus. One of the key tasks of the team is to assist its customers—any personnel from within MIT—in answering questions they have, for example, to create reports. The warehouse contains around 1TB of data spread across 3K tables aproximately.

A typical customer of the warehouse will present a question, for whith the members will need to find relevant tables manually. They create a view that is accessed by the customer to solve the question at hand. We describe next some of the common use cases we have found:

**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, year.

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

We deployed Data Civilizer as a .service and allowed the users to interact with the system through the graphical interface. Data Civilizer was able to .

## Merck

Merck is a big pharmaceutical company that manages large vol- umes of data spread across around 4K databases, plus several data lakes. The use cases are varied, however there is a common case. 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 cu- rated databases, such as PubChem [?], Chembl [?] or Drugbank [?]. We describe two common use cases that occur in this context:

**Identify entities.** One single chemical entity may be referred to with different identifier format in different databases. Chemical identifiers have been a subject of research in the bioinformatics community: multiple different formats have been proposed with different properties desirable according to the scenario. Data Civi- lizer helps by mapping the multiple representations of the identifiers, therefore facilitating the identification of entities across multiple databases, public and internal.

**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.

# CONCLUSION AND FUTURE WORK

# REFERENCES

1. Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. 24(4):557–581, 2015.
2. 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.
3. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In The semantic web, pages 722–735, 2007.
4. K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In SIGMOD, pages 1247–1250, 2008.
5. A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In SIGMOD, pages 445–456, 2014.
6. 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.
7. 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.
8. 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.
9. A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Eng. Bull., 18(2):3–18, 1995.
10. 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.
11. 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.
12. 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.
13. 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.
14. F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In WWW, pages 697–706, 2007.