Technical Report:
An Overview of Data Integration and Preparation

ElKindi Rezig
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Mike Cafarella*
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Vijay Gadepally
Lincoln Laboratory
Massachusetts Institute of Technology

May 2020

*Mike Cafarella is also with the Department of Computer Science and Engineering at the University of Michigan

Abstract

AI application developers typically begin with a dataset of interest and a vision of the end analytic or insight they wish to gain from the data at hand. Although these are two very important components of an AI workflow, one often spends the first few weeks (sometimes months) in the phase we refer to as data conditioning. This step typically includes tasks such as figuring out how to prepare data for analytics, dealing with inconsistencies in the dataset, and determining which algorithm (or set of algorithms) will be best suited for the application. Larger, faster, and messier datasets such as those from Internet of Things sensors, medical devices or autonomous vehicles only amplify these issues. These challenges, often referred to as the three Vs (volume, velocity, variety) of Big Data, require low-level tools for data management, preparation and integration. In most applications, data can come from structured and/or unstructured sources and often includes inconsistencies, formatting differences, and a lack of ground-truth labels.

In this report, we highlight a number of tools that can be used to simplify data integration and preparation steps. Specifically, we focus on data integration tools and techniques, a deep dive into an exemplar data integration tool, and a deep-dive in the evolving field of knowledge graphs. Finally, we provide readers with a list of practical steps and considerations that they can use to simplify the data integration challenge. The goal of this report is to provide readers with a view of state-of-the-art as well as practical tips that can be used by data creators that make data integration more seamless.

Contents

1 Executive Summary

Many AI application developers typically begin with a dataset of interest and a vision of the end analytic or insight they wish to gain from the data at hand. Although these are two very important components of the AI pipeline, one often spends the first few weeks (sometimes months) in the phase we refer to as data conditioning. This step typically includes tasks such as figuring out how to store data, dealing with inconsistencies in the dataset, and determining which algorithm (or set of algorithms) will be best suited for the application. Larger, faster, and messier datasets such as those from Internet of Things sensors [1], medical devices [2, 3] or autonomous vehicles [4] only amplify these issues. These challenges, often referred to as the three Vs (volume, velocity, variety) of Big Data, require low-level tools for data management and data cleaning/pre-processing. In most applications, data can come from structured and/or unstructured sources and often includes inconsistencies, formatting differences, and a lack of ground-truth labels. In practice, there is a wide diversity in the quality, readiness and usability of data [5].

It is widely reported that data scientists spend at least 80% of their time doing data integration and preparation [6]. This process involves finding relevant datasets for a particular data science task (i.e., data discovery [7]). After the data has been collected, it needs to be cleaned so that it can be used by subsequent ML models [8]. Data cleaning or data preparation refer to several data transformations to make the data reliable for future analysis. Example transformations include: (1) finding and fixing errors in the data using rules (e.g., an employee salary should not exceed $1M); (2) normalizing value representations [9]; (3) imputing missing values [10, 11]; and (4) detecting and reconciling duplicates.

This document provides a survey of the state-of-the-art in data preparation and integration. To help make concepts clearer, we also discuss a tool developed by our team called Data Civilizer [8]. Additionally, we detail the concept of Knowledge Graphs - a technique that can be used to describe structured data about a topic as a tool for organizing data with applications in data preparation, integration and beyond. Finally, this document highlights a number of practical steps, based largely on experience, that can be used to simplify data preparation and integration. A summary of these steps is given below:

1. Data Organization
    - Leverage tools for schema integration
    - Use standardized file formats and naming conventions
2. Data Quality and Discovery
    - Maintain data integrity through enforceable constraints
    - Leverage workflow management systems for testing various hypothesis on the data

- Include data version control
- Leverage data discovery tools

3. Data Privacy
    - Leverage encrypted or secure data management systems as appropriate
    - Limit data ownership
    - Avoid inadvertent data releases
    - Be aware that data aggregated may be sensitive
4. Infrastructure, Technology and Policy
    - Use data lakes as appropriate
    - Databases and files can be used together
    - Leverage federated data access
    - Access to the right talent
    - Pay attention to technology selection, software licensing and software/hardware platforms
    - Maintain an acquisition and development environment conducive to incorporating new technology advances

2 Data Integration Introduction and Challenges

Many AI application developers typically begin with a dataset of interest and a vision of the end analytic or insight they wish to gain from the data at hand. Although these are two very important components of the AI pipeline, one often spends the first few weeks (sometimes months) in the phase we refer to as data conditioning. This step typically includes tasks such as figuring out how to store data, dealing with inconsistencies in the dataset, and determining which algorithm (or set of algorithms) will be best suited for the application. Larger, faster, and messier datasets such as those from Internet of Things sensors [1], medical devices [2, 3] or autonomous vehicles [12] only amplify these issues. These challenges, often referred to as the three Vs (volume, velocity, variety) of Big Data, require low-level tools for data management, data integration [13] and data preparation (this includes data cleaning/pre-processing). In most applications, data can come from structured and/or unstructured sources and often includes inconsistencies, formatting differences, and a lack of ground-truth labels. One interesting way to think about data preparedness is through concept of data readiness levels as outlined in [5] which is similar to the idea of technology readiness levels (TRL) [14].

It is widely reported that data scientists spend at least 80% of their time doing data integration and preparation [6, 15]. This process involves finding relevant datasets for a particular data science task (i.e., data discovery [7]). After the data has been collected, it needs to be cleaned so that it can be used by subsequent ML models [8]. Data cleaning or data preparation refer to several data transformations to make the data reliable for future analysis. Example transformations include: (1) finding and fixing errors in the data using rules (e.g., an employee salary should not exceed $1M); (2) normalizing value representations [9]; (3) imputing missing values [10, 11]; and (4) detecting and reconciling duplicates.

At a high level, the concept of data integration and preparation is the effort required to go from raw sensor data to information that can be used in further processing steps (see the Artificial Intelligence architecture provided in [16]). Sometimes this phase is also referred to as data wrangling. Typically, each of these data integration and preparation tasks can be cumbersome, require significant domain knowledge, and represent a significant hurdle in developing an end-to-end application. Many of the recent algorithmic advances have, in fact, occurred in areas where "prepared" data can be found. For example, advances in image classification were largely driven by the availability of the ImageNet dataset [17], advances in handwriting recognition by the Modified National Institute of Standards and Technology (MNIST) dataset [18], and advances in video recognition by the Moments in Time dataset monfort2018moments. Other popular datasets such as CIFAR-10 [19], and Internet packet capture traces [20, 21, 22] have played important roles in advancing certain classes of algorithms and genres of applications. Data integration and preparation is often a first step in getting one's data into such a form.

There are a number of research efforts and organizations aiming to reduce

the data integration and preparation barrier to entry. For example, there are techniques to do spectral fingerprinting of datasets to look for outliers [23], techniques for modelling the background of datasets [24], and techniques for organizing data into federated data management systems [25, 26, 27, 28]. As examples of end-to-end data applications, [29] describes an oceanographic data use-case and [30] describes a medical use-case.

In this report, we highlight a number of tools that can be used to simplify data integration and preparation steps. Specifically, we focus on data integration tools and techniques, a deep dive into an exemplar data integration tool, and a deep-dive in the evolving field of knowledge graphs. Finally, we provide readers with a list of practical steps they can use to simplify the data integration challenge. The goal of this report is to provide readers with a view of state-of-the-art as well as practical tips that can be used by data creators that make data integration more seamless.

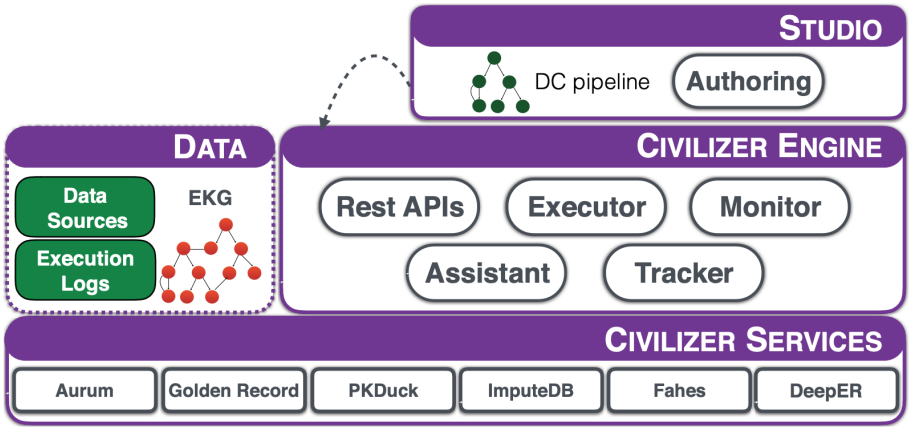

Figure 1: *Data Civilizer architecture*

3 Tools and Techniques

3.1 Data Integration Challenges

It is widely reported that data scientists spend at least 80% of their time doing data integration and preparation [6]. This process involves finding relevant datasets for a particular data science task (i.e., data discovery [7]). After the data has been collected, it needs to be cleaned so that it can be used by subsequent ML models [8]. Data cleaning or data preparation refer to several data transformations to make the data reliable for future analysis. Example transformations include: (1) finding and fixing errors in the data using rules (e.g., an employee salary should not exceed $1M); (2) normalizing value representations [9]; (3) imputing missing values [10, 11]; and (4) detecting and reconciling duplicates,

3.2 Data Integration Tasks

As part of our Data Civilizer effort [6] we have developed modules that deal with the most common data integration tasks. Figure 1 illustrates the architecture of *Data Civilizer*. Users author their data integration workflows using the Studio. The building blocks of the workflows are pre-packaged data integration modules that were designed to target the most common pain points in data integration. The Engine then runs the workflow.

We briefly describe each module and the task it addresses.

Data Discovery: First off, data scientists need to find relevant data tables from disparate data sources (e.g., data lake) that they could use for a particular task (e.g., find customers who have been making timely payments over the past 5 years). This step is referred to as *Data Discovery*. Data Civilizer includes a module, *Aurum* [7], that allows users to find relevant data tables given various

queries (e.g., is there a PK-FK relationship between the customers table and the payment history?).

Data Preparation: This step, also known as Data Cleaning, encompasses all the data transformations that are needed to bring the data to a "usable" format. Examples of data preparation include data standardization (e.g., use CS instead of Computer Sci. or Computer S.) and filling in missing values. We now present some of the data preparation modules we have developed in *Data Civilizer*.

- PKDuck [9] performs approximate string joins to map abbreviations with their full forms in a given set of table columns. PKDuck helps standardize values representation for downstream processing by replacing the full string forms with their corresponding abbreviations.
- ImputeDB [10] performs query-time missing values imputation. Given a query, ImputeDB generates a query plan that includes imputation operators to efficiently impute missing numerical values using any plugable statistical imputation technique.
- Fahes [11] detects disguised missing values (DMVs) in the data using outliers and inliers detection techniques for categorical and numerical data. Fahes replaces DMVs with NULL in each input table.
- DeepER [31] performs entity resolution using deep learning methods. Specifically, DeepER benefits from pre-trained word embeddings and user-provided training data to detect duplicate records. Additionally, DeepER makes use of blocking to reduce the number of record comparisons.
- Aurum [7] helps navigating a large collection of tables to perform data discovery by building an Enterprise Knowledge Graph (EKG) that encodes relationships between those tables. Users can query the EKG to answer various data discovery questions (e.g., joinable tables).
- A Golden Record [32] module was developed to consolidate duplicates into a single record (golden record). Before attempting to fuse the duplicate records, this module first standardizes value representations by learning candidate transformations from the data. Those transformations are then presented to the user for validation.

We now move on to the latest version of *Data Civilizer*, which, in addition to the aforementioned data integration features, includes major improvements geared towards making the jobs of data scientists easier.

4 Bridging the Gap between Data Preparation and Data Analytics

Data scientists spend the bulk of their time cleaning and refining data workflows to answer various analytical questions. Even the most simple tasks require using a collection of tools to clean, transform and then analyze the data. When a machine learning model does not produce accurate results, it is due to (1) raw data not prepared correctly (e.g., missing values); or (2) the model needs to be tuned (e.g. fine-tuning of the model's hyperparamters).While there are many efforts to address those two problems independently, there is currently no system that addresses both of them holistically. Users need to be able to iterate between data preparation and fine-tuning their machine learning models in one workflow system. We worked with scientists at the Massachusetts General Hospital (MGH), one of the largest hospitals in the US, to accelerate their workflow development process. Scientists at MGH spend most of their time building and refining data pipelines that involve extensive data preparation and model tuning. Through our interaction, we pinpointed the following hurdles that stand in the way of fast development of data science pipelines (in the sequel, we use the words "pipeline" and "workflow" interchangeably).

Decoupling Data Cleaning and Machine Learning: When it comes to building complex end-to-end data science pipelines, data cleaning is often the elephant in the room. It is estimated that data scientists spend most of their time cleaning and pre-processing raw data before being able to analyze it. While there are a few emerging machine learning frameworks [33, 34, 35], they fall short when it comes to data cleaning support. There is currently no interactive end-to-end framework that walks users from the data preparation step to training and running machine learning models.

Coding Overhead: In larger organizations, it is typically the case that several scientists/engineers write scripts that deal with different parts of the data science pipeline. While many data science toolkits and libraries (e.g., scikit-learn) have gained a wide adoption amongst data scientists, they are only meant to build standalone components and hence are not well-suited to building and maintaining pipelines involving a wide variety of tools and datasets. As a result, scientists have to write code to build and maintain data pipelines and update the code whenever they need to refine them. Because building data pipelines is a trial-and-error process, maintaining scripts hardwired for specific pipelines is time-consuming. Moreover, the effort required to try out different pipelines typically limits the exploration space.

Debugging Pipelines: When building a pipeline involving different modules and datasets, it is typical that the final output data does not look right. This is typically due to (1) a problem in the modules (e.g., bug, bad parameters); or (2) the input data to the modules was not good enough to produce reasonable results (e.g., missing values). The latter case is hard to debug using current debuggers that focus mainly on code, i.e., users have to dump and inspect intermediate data to find where it went wrong. Since it takes hundreds of

iterations to converge to a pipeline that works well for the task at hand, a data-driven debugger can significantly decrease the time spent in this process.
Visualization: Different datasets require different types of visualizations (e.g., time series, tables). Typically, scientists visualize the data in its raw format (e.g., tables) or manually visualize the data using commodity software like Microsoft Excel. However, when building pipelines iteratively, it is daunting to seamlessly integrate visualization applications (panning, zooming) into the pipeline-building process. Moreover, users need to spend a lot of time if they elect to write custom visualizations of their datasets.

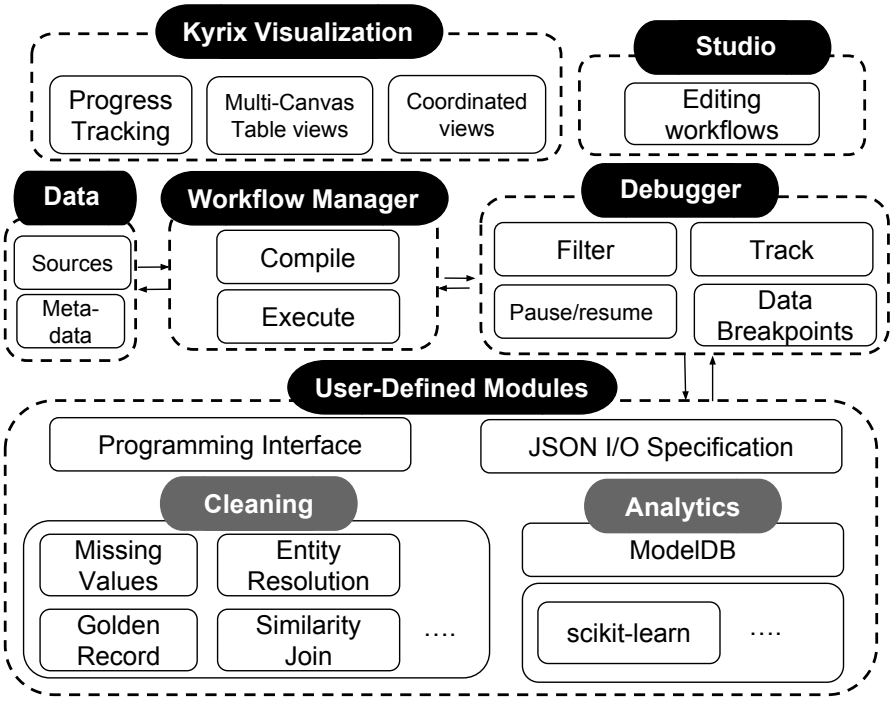

Figure 2: DC2 Architecture

There are several efforts to support data cleaning tasks [36, 37, 38], iterative machine learning workflow development [39, 34, 33, 40], and data workflow debugging [41]. Each of those efforts focuses on one aspect of the pipeline development at a time, but not all.

The previous version of Data Civilizer [6, 7, 42] focused on data discovery and cleaning using pre-defined tools. In most scenarios, users clean their data to feed it to machine learning models.

We introduce *Data Civilizer 2.0* (*DC2*, for short) to fill the gap between data cleaning and machine learning workflows and to accelerate iterative pipeline building through robust visualization and debugging capabilities. In particular,

DC2 allows integrating general-purpose data cleaning and machine learning models into workflows with minimal coding effort. The key features of *DC2* are:

- User-defined modules: In addition to a state-of-the-art cleaning and discovery toolkit that we already provide [42], users can also integrate their data cleaning and machine learning code into *DC2* workflows through a simple API implementation. Users have to simply implement a function that triggers the execution of the module they are adding.
- Debugging: *DC2* features a full-fledged debugger that assists users in debugging their pipelines at the data level and not at the code level. For instance, users can run workflows on a subset of the data, track particular records through the workflow, pause the pipeline execution to inspect output produced so far, and so on.
- Visualization: At the core of *DC2* is a component that allows users to easily implement their own visualizations to better inspect the output of the pipeline's components. We have pre-packaged a few visualizations such as progress bars for arbitrary services, coordinated table views, etc.

4.1 System Architecture

We provide a high-level description of the *DC2* architecture (Figure 2) and details are discussed in the subsequent subsections. *DC2* includes three core components: (1) *User-Defined Modules* cover required functionalities to support plugging-in existing user-defined modules into the workflow system (Section 4.1.1); (2) *Debugger* which includes a set of operations to do data-driven debugging of pipelines (Section 4.1.4) and; (3) *Visualization* abstractions to facilitate building scalable visualization applications to inspect the data produced at different stages of the pipeline (Section 4.1.3). Users interact with *DC2* using the *DC2* Studio, which is a front-end Web GUI interface to author and monitor pipelines.

4.1.1 User-defined Modules

Users can plug-in any of their existing code into a *DC2* workflow. Because cleaning and machine learning tools can vary widely, *DC2* features a programming interface that is abstract enough to cover any data cleaning or machine learning module.

Module Specification. In order to specify a new module in *DC2*, users must (1) implement a *module execution* function (executeService) using the *DC2* Python API; (2) load the module into *DC2* by specifying its *entry point file*, i.e, the file that contains the implementation of the *module execution* function; and (3) write a JSON file to list the parameters the module requires for execution.

Pipeline Execution. Service execution happens in two phases: (1) Studio generates a JSON object containing the authored workflow, which includes: module names, parameters and the connections between modules. This JSON

object is then passed to the backend (workflow manager in Figure 2) to run the workflow and; (2) every module produces a JSON object containing the path of output CSV files which are then passed to the next module in the workflow. All the *DC2* modules use a "table-in, table-out" principle, i.e., input and output of all modules is a table. In case the module fails to run, an error code is sent back to *DC2* and the pipeline execution is stopped.

executeService: The module execution function (*executeService*) takes as argument the JSON file generated from the *DC2* studio. This JSON file contains the parameter values as specified from the studio for the individual modules as well as the authored workflow. Every module (1) reads a set of CSV input files; (2) writes a set of CSV output files; and (3) might use metadata files if specified as an argument.

Every module can produce various output streams. We separate them into: *output* and *metadata*. Files produced under the *output* stream are passed on to any successor modules in the pipeline while files in the *metadata* stream are just meant to serve as "logs" that users can inspect to debug the module. For instance, a similarity-based deduplication module can produce an output stream containing the duduplicated tuples and a metadata stream that includes the similarity scores between pairs of tuples that were marked as duplicates. Each module has to produce a JSON file (output JSON) that specifies which files are produced as *output* or *metadata*.

I/O Specification. Every *DC2* module is associated with a JSON file (input JSON) containing the list of parameters the module expects and their type. Additionally, the input JSON contains the module metadata (e.g., module name, module file path). *DC2* Studio needs this specification to load the module into the GUI (e.g., if a module expects two parameters, two input fields are created in the GUI for that module).

4.1.2 Managing Machine Learning Models

DC2 supports adding machine learning models in the workflow. We integrated ModelDB [43] into *DC2* to offer first-class support for machine learning model development. ModelDB supports the widely used *scikit-learn* library in Python. Users who include machine learning modules in the pipeline can (1) track the models on defined metrics (e.g., RMSE, F1 score); (2) implement the ModelDB API to manage models built using any machine learning environment; (3) query models' metadata and metrics through the frontend; and (4) track every run of the model and its associated hyperparameters and metrics.

Moreover, we have implemented a generalization of ModelDB to track metrics in any user-defined module through a light API. The *DCMetric* class contains the following methods:

- *DCMetric(metric_name):* constructor which takes the name of the metric as a string (e.g., f1 score).
- *setValue(value):* sets the metric value. The metric can be set multiple times per run but only the final set value is exposed in *DC2* Studio.

- *DC.register(metric):* the defined metric object is registered through this function. Registration is required so the metric is surfaced in the studio.

The following is an example code snippet to track a metric "f1". First, the metric is defined (line 1). Then, the metric value is set (line 2). The metric value is finally reported to *DC2* (line 3).

```
1 DCMetric metric_f1 = new DCMetric("f1")
2 metric_f1.setValue(f1score)
3 DC.register_metric(metric_f1)
```

4.1.3 Visualization

MGH datasets are massive. For instance, the one we use in this use case is 30TB. Because we wanted to enable interactive visualizations at scale, we integrated *Kyrix* [44], a state-of-the-art visualization system for massive datasets into *DC2*. With Kyrix, users can write simple code to build intuitive visualization applications that support panning and zooming. The MGH scientists we worked with confirmed that visualization is a key component to make it easier for them to inspect their datasets. While users can write their own visualization applications using the Kyrix API, *DC2* comes with a few generic visualization applications: (1) Progress reporting: services report their progress periodically to the Studio through a progress bar; (2) Multi-Canvas Table Views (MCTV): users can click on arcs interconnecting modules on the pipeline to visually inspect the intermediate records passing between the modules and run queries on them (e.g., filter based on predicate); and (3) Coordinated Views: in the MCTV, when users select a record in one canvas, other records are selected on other canvases based on a user-defined function (e.g., provenance, records sharing same key). *DC2* comes with an API for easy integration of Kyrix visualization applications in the *DC2* Studio (e.g., show a visualization application after clicking on a particular module).

4.1.4 Debugging Suite

We have seen pipelines that run for hours, so the goal of the *DC2* debugger is to catch data-related anomalies (e.g. input data is malformed in one of the modules) early in the workflow execution, so that "bad" data is not passed to downstream processing. *DC2* features a set of human-friendly debugging operations to assist users in debugging their pipelines. We implement a GDB-like debugger that is data-driven. Users can add breakpoints by specifying a record or a set of records that satisfy predicates. Pipeline execution is paused upon reaching a breakpoint so that users can inspect visually what is going on so far in the pipeline. The following are the key debugging operations that *DC2* provides.

- **filter**: while building a data pipeline, users typically experiment with smaller datasets before testing their pipelines on the entirety of the data.

The **filter** operation allows users to specify a set of predicates to extract smaller subsets from the input datasets. For instance, if the filter is *City = "Chicago"*, then, only records with City value of "Chicago" will be passed as input to the respective module.

- **track**: an important operation when refining pipelines is to be able to track a set of records to make sure the pipeline is working as expected. Users can specify filters to track records in the pipeline (e.g., track records whose *City* attribute value is "Chicago"). Whenever a record satisfies the defined filter, it is added to a *tracking file* which contains (1) the attribute values of the record before and after going through the module; and (2) information related to the module that produced the record (e.g., name of the module, list of parameter values).
- **breakpoints**: users can specify breakpoints in the pipeline using filters. Whenever a record satisfies the filter, the execution is paused to allow the user to inspect the record at the breakpoint. Users can then manually resume the execution.
- **pause/resume**: this is a way for users to pause/resume the execution from the Studio. This functionality is implemented using breakpoints (more details in Sections 4.1.5 and 4.1.6). This operation is useful when users only want a certain module to run for a limited period of time (e.g. pause after 5 seconds). When users inspect the output and validate it, then they can resume the execution.

4.1.5 Manual Breakpoints

Data breakpoints serve as "inspection" points in the pipeline, i.e., they are used to inspect records of interest. For instance, in a deduplication module, if users notice that records whose "City" value is "Chicago" are always incorrectly deduplicated, they can add a breakpoint on records that meet the filter *City* = "Chicago", then the pipeline execution is paused whenever a record that meets the filter is encountered. We provide an API to allow users to programmatically define functions to set data-driven breakpoints. Those functions are used by the *DC2* Studio to allow users to interactively set breakpoints on records that satisfy a given user-provided filter. Three key functions need to be implemented in the *entry point file* (file containing the *DC2* API implementation) to enable manual breakpoints: (1) *setBreakpoint* which takes as argument a filter (e.g., *City* = "*Chicago*"); (2) *pause* to pause the execution when a record satisfying the filter is encountered; and (3) *resume* to resume the execution after the user has inspected the records on the breakpoint.

4.1.6 Automatic Breakpoints

In some cases, implementing the API to enable manual breakpoints can be time-consuming. To address this hurdle, *DC2* can create breakpoints in modules automatically (i.e., without requiring users to implement an API). This is done by partitioning the input data (of the module) into different subsets and running

the module with each partition. The goal is to be able to detect errors in the output of the module run with fewer partitions than with the entirety of the data. For instance, when running a classification module (with an already trained model), users might want to inspect the output for every 10% of the input data which results in nine breakpoints, i.e., output is shown after 10%, then after 20%, and so on. Additionally, the classification label of a given record does not change whether we run the model with the entire data or only a partition. If users detect misclassified records with a run using 20% of the input data, then, there is no reason to run the module for the remaining 80% records. Moreover, users can specify predicates to create partitions (blocking). For instance, "City = *" would create partitions (or blocks) where records in the same partition share the same value of the "City" attribute. Users can create automatic breakpoints from the *DC2* studio.

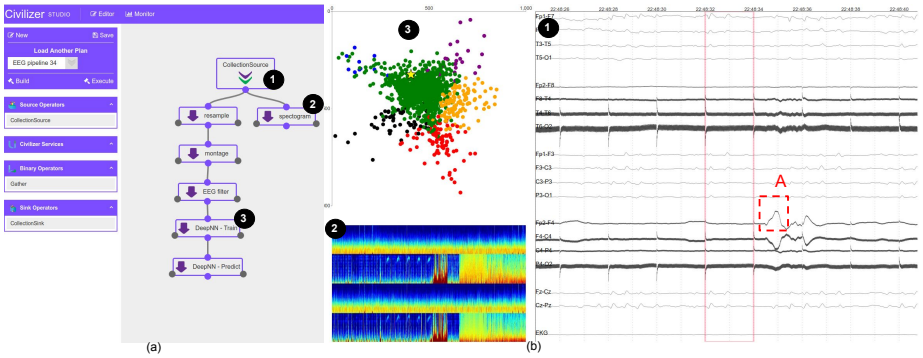

Figure 3: (a) EEG pipeline example. (b) Visualization of numbered components.

5 Data Civilizer Use Case

We used *DC2* in a real-world a medical use case with a group of scientists at MGH studying brain activity data captured using electroencephalography (EEG). Figure 3(a) illustrates an example pipeline to clean the EEG data before running it through a machine learning model. In Figure 3(a), each numbered module in the pipeline has its corresponding visualization in Figure 3(b) (e.g., module numbered 1 corresponds to raw data input).

Study. Scientists at MGH start with a study goal (e.g., early detection of seizures using EEG data), and then prepare the relevant datasets using cleaning modules. They then apply machine learning models to perform a prediction task. In the case of this use case, they want to predict seizure likelihood given EEG labeled segments. This process is iterative in nature and it takes several iterations to converge to a "good" data pipeline. We helped the MGH scientists clean and then analyze the EEG data using machine learning models. We will walk the audience through how *DC2* was used to help quickly design and execute data pipelines to carry out the study at hand.

Dataset. The EEG dataset pertains to over 2,500 patients and contains 350 million EEG segments. The total dataset size is around 30TB. Active learning is employed to iteratively acquire more and more labeled EEG segments as described in the scenario below.

Scenario. The workflow scenario goes as follows: (1) Raw EEG data is cleaned. In addition to the cleaning toolkit that comes with *DC2*, we plugged the cleaning tools MGH scientists use to clean the data into *DC2* as user-defined modules. An example cleaning task is to remove high-frequency signals (e.g., area A in Figure 3(b)); (2) Using the visualization component of *DC2*, the specialists interactively explore the 30T EEG data and then label the EEG segments based on their domain knowledge; (3) After acquiring a set of manually labeled segments, a label propagation algorithm, as a user-defined component of *DC2*, automatically propagates labels to the nearby segments of the existing labeled segments; (4) A deep learning model is then learned using part of the labeled segments as training set. During this process, the *DC2* debugger is fully explored to tune the hyper-parameters and the network structures; (5) Active learning is then conducted to improve the quality of the automatically acquired labels. First, the labeled segments out of the training set are classified by the learned model. Then using the ModelDB component of *DC2* the 2000 segments are efficiently extracted where the neural net had highest confidence but disagreed with the labels; (6) These segments are then fed back into the visualization component for the domain experts to decide whether they need to update their labels (go back to step 3) or review the cleaning step (go back to step 1). This iterative process proceeds until the neural net reaches a satisfactory classification accuracy.

6 Knowledge Graphs

6.1 Background and Terminology

Knowledge Graphs — sometimes called a Knowledge Network or structured Knowledge Bases — are a relatively novel way to describe structured data about a particular topic. Examples include Wikidata ([45]), DBpedia ([46]), the Google Knowledge Graph ([47]), UniProt ([48]), MusicBrainz ([49]), GeoNames ([50]), and many others ([51, 52, 53]). They have had slightly different definitions over the years. For the sake of this document, we think of a Knowledge Graph as a data resource in which:

- There exists a set of *unique entities* that correspond to real-world objects. For example, entity Q76 represents Barack Obama in the Wikidata knowledge network, and entity Q6279 represents Joe Biden. Different knowledge graphs make different decisions about what entities should be contained. For example, there is a Joe Biden entity in Wikidata, but not in the MusicBrainz knowledge network, which specializes in recorded music[1]. These can be thought of as the nodes in the knowledge graph.

- There are *unique properties* that describe a relationship between an entity and a data value. For example, Wikidata property P19 describes the *place of birth* relationship. For most knowledge graphs, properties are expressed with two values, at least one of which must be an entity. For example, the *place of birth* property describes a relationship between two entities in the knowledge graph, one of which is usually a person, and the other usually a location. In contrast, Wikidata's property P569 (*date of birth*) usually describes a relationship between a single entity and a data value. These can be thought of as potential edge labels in the knowledge graph.

- By combining entities, properties, and data values, the knowledge graph asserts a large number of true facts about real-world objects. For example, Wikidata states that (Q76, P26, Q13133) is true. That is, Barack Obama (Q76) *has spouse* (P26) of Michelle Obama (Q13133). These triples can be thought of as concrete labeled edges in the knowledge graph.

Our definition for a knowledge graph places it outside the ambiguity and reference problems often caused by natural language. For example, Wikidata properties P19 (*place of birth*) and P569 (*date of birth*) could both be described in English by the phrase "born in," but this point is irrelevant for the knowledge graph's correctness. There are two real-world and distinct relationships, so there are two distinct properties in Wikidata. Some applications, such as voice assistant question answering, might have to manage linguistic ambiguity, but that challenge is the responsibility of the application, not the knowledge graph.

[1] Because Barack Obama recorded the audiobooks for his self-authored books, he appears in both Wikidata and MusicBrainz. His MB identifier is `0de4d19f-05c8-4562-a3c0-7abdc144f1d5`.

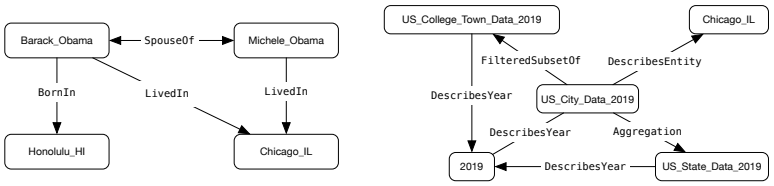

Figure 4: On the left, a fraction of a general-purpose knowledge network. On the right, a fraction of an Economics-specific knowledge network.

There is nothing preventing us from using KGs to contain entire datasets rather than single scalar values. For example, it could make sense to create an entity to model a well-known GDP dataset for practicing economists, which has multiple columns and rows; or a set of vessels when managing a fleet. Indeed, Wikipedia already does something similar when its entities contain datasets, as in the case of the Wikipedia entity `Economy_of_the_United_States`, which contains annual stats for GDP, inflation, unemployment, and so on.

Figure 4 shows two small example knowledge graphs. The figure on the left shows a portion of a general-purpose knowledge graph. The edge labels appear in many edges in the graph. The figure on the right shows an Economics-specific knowledge graph.

Our definition is not entirely consistent with all of the academic literature. There are academic knowledge graphs that could potentially create a node for every distinct noun phrase in a text, even if they refer to identical real-world objects, such as VerbKB ([54]). However, our definition is consistent with the major deployed knowledge graphs, such as Wikidata, DBpedia, and many others[2].

6.2 Graph Databases

It is useful to distinguish between a *knowledge graph* and *graph database software*. Graph database systems, such as Neo4j [59] or Amazon Neptune [60] offer query and update services for graph-oriented datasets, much like relational database systems do for traditional relational datasets. Just as traditional database software like Oracle offers the SQL relational query language, graph databases will offer a query language tailored for graphs. The most popular language is probably SPARQL, although Neo4j's Cypher language is both well-known and especially relevant when processing knowledge graphs.

Like SQL, graph query languages enable the user to ask precise and interesting analytical questions, but most users do not have the training needed to

[2]Linguistically-driven networks like VerbKB are useful in some cases. For example, Biperpedia is a "best effort" data resource extracted from text that can help search applications ([55]). Also, there is a growing literature in the NLP area that attempts to use text directly for question answering ([56, 57, 58]), without producing a knowledge network at all, but these efforts today are primarily still in the research sphere and it is unclear how to extend them to non-question-answering applications.

write them. For example, here is a simple query (drawn from the World Wide Web organization's SPARQL tutorial) that obtains a list of 50 spacecraft from a knowledge graph:

```
PREFIX space: <http://purl.org/net/schemas/space/>
SELECT ?craft
{
  ?craft a space:Spacecraft
}
LIMIT 50
```

It is possible to understand this code, but writing even this simple query will require user training.

Graph databases are designed to store a wide variety of graph-structured information, not just knowledge graph data. They are also useful for storing social network data, hyperlink data, bioinformatics data, and other kinds. Indeed, most applications of graph databases do not involve knowledge graphs. Knowledge graphs are usually of fairly modest size compared to other kinds of graph-structured data. Consider that as of this writing, Facebook has about 2.5 billion users in its social graph, while Wikidata has only 81 million entities in its knowledge graph.

Graph database systems and knowledge graphs are sometimes confused, but they play different roles. A knowledge graph is a certain kind of dataset, while a graph database system is a piece of software. Although they are often useful together, one cannot be substituted for the other.

6.3 Application Examples

There has been a substantial amount of research into methods for construct knowledge graphs, such as information extraction [61, 51, 62, 40, 63, 52, 64, 65] and crowdsourcing [66, 67, 68, 69, 68].

In contrast, there has been relatively little research into the knowledge applications built on top of KGs, outside a few important but narrow use cases, such as personal assistants [70] and precision medicine [71].

The best-known industrial use cases of knowledge graphs are *structured web search* (see Figure 5 and *voice assistants* such as Siri, the Google Assistant, and Amazon's Alexa product. In both types of systems, a knowledge graph serves as a generic multitopic database to produce user answers.

It is perhaps useful to consider the rough pipeline of steps that take place in a voice assistant to understand the role that a knowledge graph plays. Imagine that a user has just asked the Google Assistant, "Where was Barack Obama born?" Conventional engineering wisdom suggests the voice assistant goes through the following steps:

1. The device records user speech saying, "Where was Barack Obama born?"
2. Voice assistant software uses a speech-to-text system to translate the utterance into a textual string

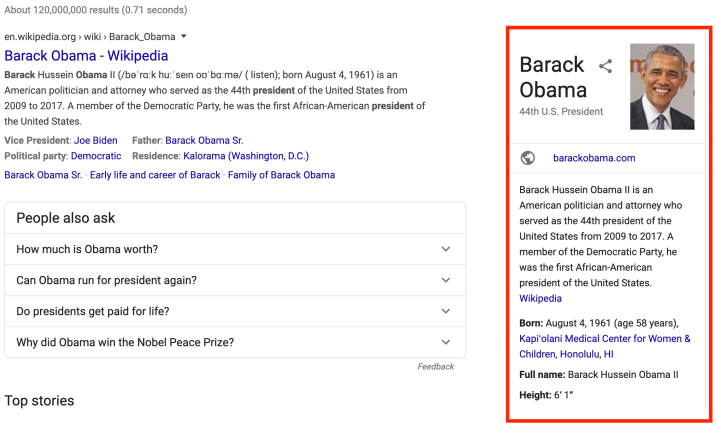

Figure 5: A Google search yields the traditional list of hyperlinks at the left, and often a knowledge graph-derived structured results on the right-hand side. In this case, **born**, **Full name**, and **Height** are all properties of the **Barack Obama** entity in the KG.

3. The voice assistant applies natural language processing methods to the text string in order to translate it into a structured graph query in SPARQL or something similar.

4. A graph database system that stores the knowledge graph will execute the SPARQL query and return a result.

5. The voice assistant will turn the result into audio that can be played to the user.

A large and high-quality knowledge graph is necessary, but not sufficient, for a successful voice assistant. The voice assistant can only answer questions that have answers in the knowledge graph, which potentially explains Google's private efforts to create a very large one [47].

6.4 Wikidata

Wikidata is a widely-used open-source knowledge graph. Wikidata was founded in 2012 and has since become the largest and most successful of the public general-interest knowledge graphs. Wikidata is unusual in being both very large and very transparent. In contrast, most other knowledge graphs are either small research projects or large private endeavors. It is a volunteer nonprofit effort, with funding donated from various charitable and scientific organizations.

As of this writing, Wikidata contains structured data about roughly 82M entities. An entity is any general-interest real-world object, such as Barack Obama (https://www.wikidata.org/wiki/Q76), the city of Chicago (https://www.wikidata.org/wiki/Q1297), the Federal Reserve (https://www.wikidata.

org/wiki/Q53536), or the US Navy (https://www.wikidata.org/wiki/Q11220). Wikidata contains approximately 1 billion factual statements. Roughly half of the entities (40M) contain 10 or more factual statements.

Wikidata is remarkable in its ability to simultaneously obtain large scale and data quality. Not only is the factual precision high, the set of properties and types — Wikidata's version of a schema — is remarkably consistent across the dataset. For example, there is a single property *spouse* that is widely used; there are few or no unnecessary duplicates of the concept. This might not sound like a major achievement, but consider much overlap and duplication exists in all the relational database schemas in an organization. Even a simple concept like *employee* can be modeled in many different ways in different databases; if the organization wants to run a query that examines *all* of the employees, it often must perform a long and expensive data integration process. Wikidata is somehow able to avoid most of these modeling and integration problems, even though its dataset and topic diversity are vast.

It is not entirely clear which Wikidata practices are most responsible for their success, but here are a few notable ones:

1. Almost any user can contribute a new fact to Wikidata, without any scrutiny before it is added and made publicly visible. However, they are limited to using properties (e.g., *spouse* or *sibling* or *date of birth*) that already exist.

2. Any user who wants to add a novel property must submit the request for approval before the property can be used in a new fact. The request is reviewed by a small panel of senior Wikidata editors.

3. The software interface for adding a novel fact includes aggressive auto-suggest, which encourages users to select properties from the preexisting set.

4. The Wikidata software retains full version history of all facts, so it is easy to undo any unwanted changes.

5. Editors have tools that allow them to quickly review large numbers of factual additions, letting them examine and potentially undo fact insertions with just 1-2 keystrokes.

6. The system, somewhat surprisingly, does not use much in the way of machine learning-style probabilistic machinery.

Unfortunately, there is no commercial product that allows a user to easily replicate the Wikidata process when building a novel knowledge graph. However, it is clearly an exciting and relevant example that can potentially be emulated.

6.5 Future Research

By massively expanding the availability of well-administered structured data, and by allowing anyone (not just database administrators) to improve that structured data, knowledge graphs promise a revolution in the kinds of data-powered applications that can be constructed. Everyday users could stop issuing simple document ranking queries; instead, with the same trivial amount of work, they could easily ask detailed questions about the world and receive a customized data-driven answer. Professional data analysts could easily focus on novel topics, not just the databases that have been cleaned and prepped for them. Visualizations in a report or a news article could be tied to their backing data stores, allowing any reader to explore a data point that seems suspicious.

More concretely, consider:

- An knowledge graph that contains virus-related scientific research, when combined with an KG-powered application, could allow a user to quickly identify the researchers who have authored the most papers that mention both remdesivir and a filovirus.
- An knowledge graph that describes naval vessels, their crew (**terminology?**), and their equipment could power an application that finds repair opportunities.

This exciting future has arrived in small glimpses: voice assistants, and search engines that give direct answers instead of forcing users to trawl through documents. Unfortunately, knowledge graphs do not exist for many topics, and KG-powered applications exist for fewer. The exciting knowledge graph future has been frustratingly slow to arrive. We need domain-independent infrastructure to hurry it along.

Knowledge Graph Construction — One core reason is that constructing a high-quality KG itself is difficult and time-consuming. Google has been developing its private knowledge graph for over a decade; and even though the Wikidata project is one of the best KGs available, it is almost eight years old but still has quality and coverage challenges. These are large projects, and so perhaps have necessarily taken a long time. But in past interviews with KG researchers, we found that even a small new knowledge graph project typically takes 12 months, even with highly-trained Ph.D.-level engineers.

Knowledge Applications — The pain associated with building applications for knowledge graphs is even more acute. Only a tiny number of the most resource-rich organizations have been able to pursue them. Consider that building an KG-powered application entails:

- Writing complicated data acquisition code that clumsily exports data from a KG, then reimports it. Worse, this code simply breaks when the KG changes, even when it *improves*.

- Fixing bugs that can hide in many more places than in traditional software: bugs can lie in source code in the application itself, or incorrect inputs, or incorrect shipped data from collaborators.

- Shipping massive datasets to collaborators, and finding data quality error induced by downstream software.

Moreover, because KGs and their applications have a symbiotic relationship — better applications raise the demand for good KGs, and better KGs enable better applications — the huge cost of developing applications means that KGs develop more slowly than they otherwise might. Unsurprisingly, the recent explosion of tech company investment in improving general-purpose KGs has gone hand-in-hand with increased popularity of applications like the Google Assistant.

Research Goals — We propose to research and build infrastructure that will make KGs and applications dramatically cheaper and easier to build. We plan to do so in two ways:

- Building a Knowledge Graph Programming System, or KGPS. This is a programming language and toolset specifically designed for building KG-driven software and for debugging data quality problems more quickly and at lower cost than is common today.

- Building and rapidly shipping KGs for several applications, in order to gain experience and test integration with the KGPS. We will initially target COVID-19 science, plus related applications. Better scientific understanding of the virus is an urgent social need that this KG and its applications can help address, while also serving as a testbed for our infrastructure.

We can now describe our plans for the KGPS in more detail.

6.6 The Knowledge Graph Programming System

The Knowledge Graph Programming System is a programming system that takes the KG as its core data representation. It makes KG applications easier to build and easier to improve. Its features fall into three main areas:

1. **Easier Knowledge Acquisition** — KGPS application code can use values and types that cover the "real world" just like a KG does. There is a value for `Barack Obama`, one for `Boston Red Sox`, and so on. There is a type for `Politician`, for `Cartoons`, *etc.* Unlike most programming platforms, the KGPS library is intended to have the same comprehensive coverage that a good KG does. As a result, programmers can very succinctly obtain data from a backing KG.

2. **Easier Knowledge Debugging** — Every output value that KGPS has a globally-accessible URL that carries the full history of how the value was

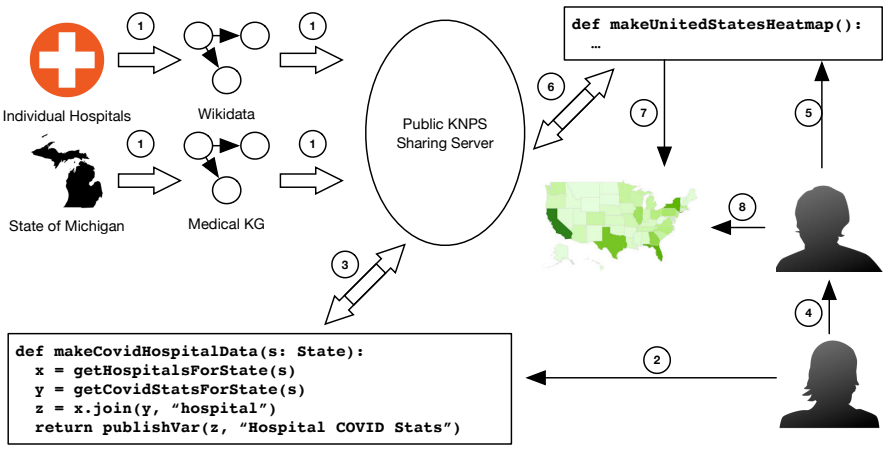

Figure 6: An example of how the **Knowledge Graph Programming System** enables efficient and high-quality knowledge applications in the case of COVID-19 analysis.

computed. Many applications today are a mixture of data and code collected from a hodgepodge of colleagues, models, and opaque institutions; as a result, debugging an incorrect output can turn into a tedious goose chase rather than an engineering exercise. In contrast, KGPS application values will automatically have full lineage information that makes them debuggable by every programmer.

3. **Easier Knowledge Sharing** — Data pipelines are everywhere in modern data science, including model training procedures, crowdsourcing updates, web crawlers, sensor readings, market updates, data cleaning methods, or just bureaucratic processes. Each of these pipeline steps can entail heavy data transfer costs. Worse, errors can be induced by pipelines entirely outside the application's control. KGPS programs will have data sharing operations as a builtin primitive, making it easier to collectively diagnose and repair knowledge applications, even if those errors are tied to errors from upstream or downstream pipelines.

Finally, the KGPS system also has one very boring quality: it is mainly Python, so data scientists and programmers can use it with little change to their daily routines.

6.7 Example Walkthrough

It is perhaps easiest to describe KGPS's behavior by showing a simple walk-through example. The following narrative discusses steps illustrated in Figure 6:

1. In **Step 1**, various hospitals over time add statistics about themselves (date of founding, number of beds, etc) to the public Wikidata Knowledge Network. The State of Michigan regularly publishes the latest per-zipcode COVID case numbers data to a public health KG. All updates to these KGs are monitored by a KGPS **sharing server** and turned into variables and values that are available to all KGPS programmers.

2. In **Step 2**, a programmer writes a short KGPS function that computes the number of COVID cases per hospital bed in each zipcode in a state, applying it to the value `Michigan`. Because all of the data is available via the KGPS server, the programmer can load it with a single unambiguous function call, rather than slinging CSV files or SQL for private databases.

3. In **Step 3**, the program runs and computes the dataset of cases-per-bed. The result is both returned to the programmer and is synchronized with the KGPS server. The KGPS server returns a URL that can be used to permanently refer to the generated dataset.

4. In **Step 4**, the programmer sends this URL to a medical data analyst at a different institution, so she can experiment with this new dataset.

5. In **Step 5**, the medical data analyst writes a short KGPS function that uses this Michigan cases-per-bed dataset to build a national heatmap of hospital usage intensity. This program refers to the dataset using the URL sent by the programmer.

6. In **Step 6**, the heatmap program runs. It needs the programmer's data in order to run, so it contacts the KGPS server and downloads the data.

7. In **Step 7**, the heatmap program completes and generates its output image. The medical data analyst examines the image and finds a surprising result in the heatmap near Ann Arbor.

8. In **Step 8**, the medical data analyst examines the entire lineage of how the heatmap was computed, including the original programmer's code, and any crowd modifications to the source KGs. She realizes that the number of hospital beds in Ann Arbor is incorrect, so she edits Wikidata to fix the problem. Because she has access to the full history of how the hospital data was computed, it is trivial to reexecute the entire chain of programs and to obtain a satisfactory heatmap.

This short example illustrates many ways KGPS can make KGs more useful and knowledge workers more productive. The programmer in Step 1 can obtain an accurate list of US hospitals in just one line of code. The medical data analyst can easily examine how her programmer friend computed the dataset she received in Step 4. She can quickly find the source of the bug she identified in Step 7's output. The medical data analyst in Step 8 can directly fix the upstream data bug in Wikidata, so that everyone can benefit from the fix.

```
1       def computeStandardBedsPerResident(h: Hospital):
2         nearbyZipcodes = Zipcode.getAll().filter(zc: zc.distance(h.zipcode) < 40)
3         nearbyPopulation = 0
4         for z in nearbyZipcodes:
5           nearbyPopulation += z.population
6         return nearbyPopulation / (h.beds - h.icuBeds)
7
8       computeStandardBedsPerResident(KNPS.get("Univ of Michigan Hospital"))
```

Figure 7: KGPS code to compute the number of residents per bed in a hospital's service area.

6.8 Language Features

We now go through each of the key KGPS features.

6.8.1 Types and Values

A core design goal of the KGPS is to make it easy for programmers to acquire, use, and update KG data. As a result, KG entities are a primitive data value, and KG-derived types are built in to the KGPS. In other words, programs are not limited to describing the world in terms of integers, strings, and sorted arrays. Rather, programmers can write code that models the real world much more directly.

As a result, programmers can easily identify sets of data to load without handling files, formats, or SQL. They can also exploit a high-quality preexisting set of schema decisions for real-world objects; these are derived from a source KG. With the KGPS value and type system, much of the data loading and management code that is done by data software can effectively be offloaded onto preexisting KG processes.

Figure 7 illustrates KGPS code that computes the number of standard hospital beds per resident offered by a particular hospital for a standard service area. On line 1, the `Hospital` type is part of the KGPS builtin class library, derived from a backing KG. This `Hospital` type has various useful fields that have been developed and populated by the KG's data curation process, so the programmer does not have to. On line 2, the programmer uses the builtin `Zipcode` type to get a list of all zipcodes, then filters them to retain only the zipcodes within 40 miles of the hospital's zipcode. The `Zipcode.distance()` function and the `Hospital.zipcode` field are all part of the standard KGPS library. Many other useful fields are also part of the KG-derived types: the `population` field on line 5, and `beds` and `hospitalBeds` on line 6. Finally, on line 8, the programmer grabs a data value that represents the University of Michigan Hospital, which is an instance of the `Hospital` type.

Broader Impact — Without the KGPS builtin values and types, the code in Figure 7 would be substantially harder to write. Programmers would have to

build a custom `Hospital` class and define its fields, build or download a `Zipcode` class and define its `distance()` function, then find data for the University of Michigan and hospital and load the data into the appropriate fields. KGPS relies on KG-derived data to do all of that tedious, but crucial, data modeling work.

Intellectual Novelty — The KGPS builtin library is unlike most programming language libraries: it is intended to cover *every topic*, so the programmer rarely has to worry whether a particular concept is part of the system; she can simply assume it exists. We pursue this by asking the developer for a tiny number of sample instances for each desired class, then exploit recent advances in knowledge graph embeddings (such as [72], [73]) to automatically build a type-specific prediction model. However, the type library cannot shed previously-defined fields or functions without potentially breaking KGPS user code; this challenge is quite similar to problems in data integration problems ([74], [75], [76]).

6.8.2 Automatic Provenance

Another core design goal of the KGPS is to make all data values easy to debug. Even today's limited knowledge applications entail a vast number of moving parts and the efforts of many different people. For example, answering a single voice assistant query might depend on a fact added to the knowledge network by a Wikidata user, a piece of voice recognition code added by an engineer on an open source recognition project, a large set of training examples contributed by users, and interface code written by the company that made the voice device. If you get a wrong answer to your voice assistant query, which of the above elements was to blame?

This crucial data debugging is in some ways similar to traditional software debugging, except that the full program execution is spread out in time, across many machines, and performed by many different participants. Without knowledge of the full history of any particular piece of data, fixing errors is not possible. Developers must perform tedious detective work and track down every possible input, or in many cases simply add more training data to the model process and hope for the best. Although there has been some recent research into systems for data debugging, it has focused on debugging a database in isolation, not data in the context of a multicomponent application ([77]).

KGPS addresses this challenge by giving *every single value* the full provenance of how it was computed. There has been a substantial amount of research in data provenance — sometimes called data lineage — in a database or reproducibility setting ([78, 79, 80, 81, 82, 83, 84]). But there is no standard programming system in which provenance plays a major role. KGPS aims to perform this service.

There is a huge number of existing provenance systems, but they tend to either require a substantial amount of developer adaptation, or are incomplete and leave some portion of the value history uncaptured. We will instead follow a pol-

icy of *automatic provenance capture* and *degrading provenance accuracy rather than coverage*. We have previously used synthesized code for data-intensive tasks to obtain higher performance in opaque user code ([85]), or to perform data manipulation operations ([86, 87]); we will follow the same strategy with provenance capture. When dealing with software components that cannot be instrumented or wrapped, we will use machine learning models to predict the likely provenance values; it might seem irresponsible to employ potentially-inaccurate provenance values, but in a debugging context they will often be quite useable.

Broader Impact — KGPS adds two features not typically seen in provenance systems. Crucially, it will compute *provenance without programmer interaction* and *across programs and storage systems*. We aim to do this via synthesized wrappers around user-written KGPS code. If successful, this approach would make provenance a practical reality for KGPS application programmers, bringing the full history of every data value to the programmer's fingertips; as a result, data debugging would be vastly easier.

Intellectual Merit — There is a huge amount of work in data provenance, but also a lack of practical systems in common practice outside a few narrow use cases. This suggests *deployable provenance* is a promising and challenging direction. Synthesizing wrappers is a known approach, although often done in a crude manner that simply logs the method call. We are unaware of existing work that pursues this degraded-accuracy approach for maximizing provenance coverage.

6.8.3 Sharing Values and Variables

Finally, KGPS will allow easy sharing of values and variables. Running a KGPS program will essentially generate its own synthetic "execution knowledge network", designed to cover the topic of its own execution history. Values in this network can be shared with others and commented-upon, just like values from Wikidata that are shared with the KGPS server in Figure 6. This will allow KGPS programs to easily add new items back to the shared knowledge network, which can then be easily reused by other developers.

6.9 Alternate Interaction Modes

The above text describes the standard programmatic interface to KGPS. But once users add code and data to the system, other interaction modes are possible. In Phase 1 we built a search engine-like interface for analysts to use, which exposes KGPS functions and data via a website. Users are able to obtain useful and interesting programmatic results with just a single line of text.

7 Practical Steps

This section outlines a number of practical steps that can simplify data integration. We organize these practical steps into:

1. Data Organization
2. Data Quality and Discovery
3. Data Privacy
4. Infrastructure, Technology and Policy

7.1 Data Organization

- **Schema Integration:** Schema integration is one of the core challenges for any organization looking to query disparate datasets. A simple example: one database of employees might contain `startdate` as a calendar date, while a second one contains `beganservice` as a year. Writing a query that uses both of these databases entails figuring out how to map `startdate` to `beganservice`. In practice, solving this seemingly-simple problem can become quite daunting and expensive.

  Most traditional schema integration tools are intended to integrate preexisting schemas with extremely high accuracy. This makes the tools suitable for important integrated queries, such as computing payroll for people in a large organization with many employee databases. These tools often come with bundled professional services. IBM and Palantir are two well-known vendors. These projects tend to be time-consuming and relatively expensive. This is usually not a viable approach for projects that aim to make all data in the organization queryable: the per-schema integration cost is simply massive when the number of schemas is large.

  There are research and startup efforts that use probabilistic methods to perform schema-specific integration at more reasonable cost. Tamr is one small but growing vendor. These can be successful, but may have unusual requirements, such as extensive employee-annotated training data.

  More recently, knowledge graphs have employed web front-end tools and social processes to produce a single large-scale integrated schema. Essentially the knowledge graph's schema is the "target"; all raw data in the world must be formatted to fit this target or it cannot be added to the knowledge graph. This is a major achievement, especially considering the relatively low financial cost. However, it has come at the cost of some flexibility. Users who cannot accept the knowledge graph's target schema (perhaps to retain backward compatibility with legacy software) simply cannot add their data to the knowledge graph. Making matters worse, there is currently no standard commercial offering to enable private knowledge graph construction.

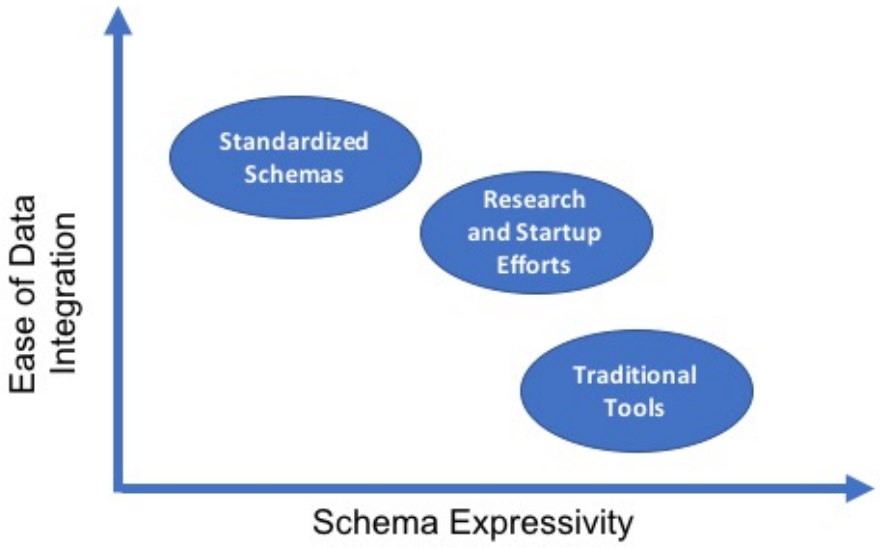

Figure 8: *High-level view of various tools for schema integration*

It may sound like promulgating standardized schemas would enable the best of all worlds, but most organizations find these difficult to implement. One way to view the success of knowledge graph projects is that they are effectively social software systems that encourage organizational agreement on a standardized schema. Although there is no commercial product available today in this area, it is worth watching for in the future. Figure 8 give a high-level view of these various tools.

- **Data Formatting and Naming:** One of the easiest ways to simplify data integration is through standardized files in human-readable formats with standardized filenames. We have observed that in many instances, a data management system can be nothing more than a shared filesystem with standardized filenames and folders. Wherever possible, parse as much data as is practical into tabular files. Try to avoid proprietary formats. If possible, use .csv (comma separated values) or even better .tsv (tab separated values) file formats. Include column labels (each column label should be unique within the file). Have row labels in the first column, such as row number or record number (each row label should be unique within the file).

  Avoid lots of tiny files and compress with zip when possible. Use hierarchical directories to keep the number of items in a directory reasonable (less than 1000 files/directory). Use well-defined directory structures to provide easy to access information about the dataset. For example:

`source/YYYY/MM/DD/hh/mm/source-YYYY-MM-DD-hh-mm-ss.tsv`
or
`YYYY/MM/DD/hh/mm/source/YYYY-MM-DD-hh-mm-ss-source.tsv`

7.2 Data Quality and Discovery

- **Data integrity:** In most cases, data becomes dirty because organizations do not invest in setting up policies that control the quality of the data *before* it is introduced to the database. It is always a good idea to think about constraints that should hold on the data at all times (e.g., employee salary must be greater than 0). This way, if the data is updated, we will make sure those updates would not violate the constraints.

  Fortunately, there is a plethora of work that addresses ways to design constraints to keep the data consistent at all times. Functional dependencies and Denial Constraints are some examples [88, 89]. Implementing those constraints could be done either at the application level, i.e., write code to enforce those rules at the application, or at the DBMS level, i.e., write triggers that call a SQL procedure. The latter has the advantage of enforcing the constraints for any number of applications interacting with the database.

- **Testing various hypotheses on the data:** It is estimated that designing data processing pipelines takes hundreds of iterations on average [90]. As the data undergoes a myriad of transformations over time, it is crucial to be able to quickly test hypotheses on the data to make sure the pipeline is up-to-date (e.g., new training data available, new updates to base tables).

  Workflow management systems can help design and test data pipelines as the data and the organizations specifications (e.g., new department) evolve. Data Civilizer is one example of such a tool [8].

- **Data version control:** Typically, data is handled by different actors in an organization (different departments, etc.). It is important to keep track of "who did what" to the data over time. This makes it easy to ask the right people about versions of the data (instead of asking everyone in the organization). This is especially important when dirty data starts to appear as a result of a recent event (e.g., someone has inadvertently made a "bad" data update). An example system design is outlined in [91].

- **Data discovery:** Most organizations have a data warehouse or a data lake where they dump historical data for future analysis. As data analytics becomes a central part in modern organizations, it is important to make the job of "data scientists" as productive as possible. A well-cited figure states that data scientists spend over 80% of their time preparing the data [8]. This leaves them with very little time to perform their analytical

tasks. One key component of data preparation is Data Discovery. The premise of data discovery is: Given a set of tables (e.g., data lake), we would like to extract tables/records of interest to our task.

7.3 Data Privacy

- **Database Technologies for Privacy:**Protecting diverse datasets, their aggregates, and subproducts may require encrypted databases such as those presented in [92, 93, 94, 95]. Essentially, these systems attempt to enforce organizational policies on data confidentiality, integrity and/or availability of data throughout the data lifecyle. An overview of the state-of-the-art in encrypted databases is given in [96].

- **Data Ownership:** Limit data ownership and access to those with a need to know. While there are algorithmic solutions for such policy checking, these are often very difficult to implement and have major issues when something goes wrong. There are new techniques proposed that may simplify the implementation of complex access control policies such as query based access control [97]; however, these approaches are still relatively new and in the research phase.

- **Inadvertent Releases Are Very Possible:** Unfortunately, ensuring data privacy is an enduring problem for which there are no easy solutions. In general, organizations can improve by minimizing data collection whenever possible, and by following good storage practices of sensitive data (such as storing data in encrypted form, and not allowing wide copies of sensitive files). These measures will help minimize the frequency of the most egregious "data spills", such as releasing an entire database of private financial information.

  However, even cautious organizations can make errors. A famous example is connected to the Netflix Prize contest of the mid-2000s. The Netflix Prize was a machine learning contest, in which Netflix challenged research teams to predict customers' movie ratings more accurately than any other team. As part of the contest, Netflix released a database of customer movie ratings, with all customer information replaced by an opaque identifier. The data had four fields: `user-identifier`, `movie-title`, `grade`, and `date-of-grade`.

  After Netflix released this data, researchers were able to identify the real names of some of the customers in this database. They examined the public rating history of users on IMDB.com, which reveals actual usernames. By looking for pairs of users who had distinctive and overlapping movie histories on both sites, they were able to tie the Netflix user identifiers to IMDB user accounts. Some of one user's Netflix rating history contained sensitive movie titles that were not part of the user's IMDB rating history, and which arguably revealed information about the user's sexual orientation.

In short: Netflix was careful to act responsibly in its data release, but still ended up enabling the revelation of private information. Technical solutions to this problem are currently being explored in the research world (differential privacy is one direction), but are generally not deployable today.

The US Census Bureau is a potentially useful model to emulate. They maintain a large amount of personal information about many Americans, which most would find extremely intrusive if it were released. This data is used by researchers and policymakers. However, the census data is believed to have never been part of an unintentional privacy breach, despite its long history[3]

In order for researchers to use much of the Census Bureau data, they must:

- Undergo a background check.
- Physically visit one of 29 Federal Statistical Research Data Center access points around the country, where they use a government-provided terminal.
- Not carry into the access point any electronic means of copying the data.

In other words, the Census Bureau has avoided Netflix's strategy of deidentification plus wide distribution. They have instead simply limited access to the data. This likely has come at some cost to researchers, but does seem to have achieved the data privacy goal.

- **Sensitivity of aggregated data:** Integration of different non-sensitive datasets may lead to sensitive outputs. For example, consider a medical scenario in which a user is trying to integrate data from *Table 1* and *Table 2*. Suppose *Table 1* consists of columns (PatientID, Patient Name, Patient Gender) and *Table 2* consists of columns (PatientID, Patient Date of Birth, Patient Age). If a researcher is interested in looking at the distribution of gender and age within the datasets, they may attempt to do a "join" on Table 1 and Table 2 using the PatientID. Very often, combining Patient Name and Date of Birth can lead to sensitive Personally Identifiable Information (PII). However, if the user instead filtered their queries to only select the pieces of data relevant from each table, the aggregated query may not lead to PII information. Similar cases exist within Department of Defense (DoD) and enterprise datasets. There are a couple of approaches to mitigate this risk. In one direction, you could run all services on a sensitive network. Thus, any data products will already be on a system or network approved for the aggregate (e.g., the Census Bureau example above). While this may seem like the easiest path forward for developers of the service, from a user perspective, this is not convenient

[3] An arguable exception: census data was used to support the internment of Japanese-Americans during the Second World War, at the direction of government officials [98].

(or feasible). Another approach may be to leverage encrypted databases [99, 100, 93, 96] such that all data products are kept encrypted and can only be viewed if certain constraints are satisfied (e.g, a filter that ensures any data being decrypted doesn't contain sensitive aggregates). These tools, largely research efforts, are worth watching and may provide an interim solution. In the long term, we believe that these approaches in conjunction with automated tools built on knowledge graphs are a promising direction. Knowledge graphs would allow Information Security Officers (ISOs) of various organizations to relatively easily enforce various security policies. For example, a policy may be that `Date of Birth` and `Patient Name` should not be aggregated on a non PII-compliant system. With a knowledge graph representation, each record in the database is tied to an entity type and this policy is easy to enforce. In such a case, the data integration tool may be able to alert users that they are likely to have a policy violation and may even be able to help them construct a policy compliant query. A good overview of the challenges is given in [101].

- **Experience with medical data access:** We collaborate with two groups at the Massachusetts General Hospital to help them manage and clean their data. Managing access to this data has to be done carefully since it is sensitive data. As collaborators, we (1) cannot copy the data to any machine and have to use the data only within the MGH servers; and (2) the data we have is de-identified. We also review any code that reads this data before deploying it. Data breaches often happen because of vulnerabilities in the code (e.g., web server assumes all requests are benign).

7.4 Infrastructure, Technology and Policy

- **Data Lakes:** A common software deployment is a so-called "data lake": a single physical location where all members of an organization place datasets, often in raw formats. The Hadoop Distributed Filesystem is a common infrastructure for data lakes, sometimes with some amount of administrative tooling. Importantly, there is usually little or no semantic integration; schemas might not even exist or be declared for all the data, let alone be integrated.

  Data lakes can serve some useful roles, but users will likely be disappointed if they hope the data lake alone will enable them to query the entire organization's data. Physically colocating the data is a potentially useful step, but semantic integration among datasets is often a much more time-consuming and expensive hurdle. Further, in many cases, physical colocation might not even be necessary for semantic integration.

  One scenario in which a data lake can be useful is when the data being uploaded is quite homogeneous. For example, it might be useful to

transmit all of the log data from a set of servers to a single location. Another is when an organization has a sophisticated data science team that is ready to build novel analyses, and simply lacks physical access to all of the organization's data.

- **Database vs. Files:** Databases and files can often be used together. Use databases if you need to quickly find particular data items (i.e., you are looking for a small number of records when compared to the entire dataset). SQL systems are good for medium size datasets that require ACID (atomicity, consistency, isolation, and durability) guarantees. NoSQL systems are good for large datasets where ACID guarantees aren't required [102]. NewSQL systems are good when you have a need for scalability and ACID compliance [103]. Scanning files in the file system can be best when reading in a majority of a dataset.

- **Federated Data Access:** Leverage a scalable data management architecture that allows the addition of new technologies such as object stores, databases and file systems. It is unlikely that any single technology solution will scale or provide efficient access to all modalities of data and federation as an architectural principle is important [25]. As an example, polystore systems (an example, BigDAWG is described in detail in [27, 104, 105]).

- **Talent:** Access to the right people. Need for data scientists, subject matter experts in addition to data users.

- **Technology Selection:** Have Top-Down technology selection. Involve management in technology selection process. Avoid products/technologies that have unknown/unreliable development team (e.g., teams for adversary nations; teams of individuals unlikely to continue maintaining software)

- **Software Licensing:** Be aware of software licensing: Certain software libraries and products have restrictive software licenses. This may limit the ability to share technology with other industry/academic partners.

- **Software and Hardware:** Avoid software/hardware products with unknown user base or non-active developer base. Reevaluate software/hardware products when the developers are acquired by other entities.

- **Incorporating new tools:** Maintain an acquisition and development environment conducive to incorporating new technology advances. Many of the challenges discussed in this document are active areas of research in the commercial and academic sector. Innovations are coming at a rapid pace and it is important that decision makes maintain an environment to acquire and deploy these technologies in an agile fashion.

8 Acknowledgements

The authors wish to thank the following individuals for their support in developing this report: Siddharth Samsi, Jeremy Kepner, Albert Reuther, Mike Stonebraker, Sam Madden.

This material is based upon work supported by the National Science Foundation under Grant No. 1636788 and by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Sulayman K Sowe, Takashi Kimata, Mianxiong Dong, and Koji Zettsu. Managing heterogeneous sensor data on a big data platform: Iot services for data-intensive science. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 295–300. IEEE, 2014.

[2] Alistair EW Johnson, Tom J Pollard, Lu Shen, H Lehman Li-wei, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3:160035, 2016.

[3] Mohammed Saeed, Mauricio Villarroel, Andrew T Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H Kyaw, Benjamin Moody, and Roger G Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database. *Critical care medicine*, 39(5):952, 2011.

[4] Vijay Gadepally, Ashok Krishnamurthy, and Umit Ozguner. A framework for estimating driver decisions near intersections. *IEEE Transactions on Intelligent Transportation Systems*, 15(2):637–646, 2013.

[5] Neil D Lawrence. Data readiness levels. *arXiv preprint arXiv:1705.02245*, 2017.

[6] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *CIDR*, 2017.

[7] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1001–1012. IEEE Computer Society, 2018.

[8] El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. Data civilizer 2.0: A holistic framework for data preparation and analytics. *PVLDB*, 12(12):1954–1957, 2019.

[9] Wenbo Tao, Dong Deng, and Michael Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.

[10] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. Query optimization for dynamic imputation. *PVLDB*, 10(11):1310–1321, 2017.

[11] Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Mourad Ouzzani, and Nan Tang. FAHES: detecting disguised missing values. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1609–1612. IEEE Computer Society, 2018.

[12] Vijay Gadepally, Arda Kurt, Ashok Krishnamurthy, and Ümit Özgüner. Driver/vehicle state estimation and detection. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 582–587. IEEE, 2011.

[13] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 2002.

[14] Brian Sauser, Dinesh Verma, Jose Ramirez-Marquez, and Ryan Gove. From trl to srl: The concept of systems readiness levels. In *Conference on Systems Engineering Research, Los Angeles, CA*, pages 1–10, 2006.

[15] Michael Stonebraker and Ihab F Ilyas. Data integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9, 2018.

[16] Vijay Gadepally, Justin Goodwin, Jeremy Kepner, Albert Reuther, Hayley Reynolds, Siddharth Samsi, Jonathan Su, and David Martinez. Ai enabling technologies: A survey. *arXiv preprint arXiv:1905.03592*, 2019.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[18] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[19] Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

[20] Vijay Gadepally, Jeremy Kepner, Lauren Milechin, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Matthew Hubbell, Micheal Houle, Micheal Jones, et al. Hyperscaling internet graph analysis with d4m on the mit supercloud. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[21] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *Proceedings of the 6th International COnference*, pages 1–12, 2010.

[22] Emily H Do and Vijay N Gadepally. Classifying anomalies for network security. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2907–2911. IEEE, 2020.

[23] Vijay Gadepally and Jeremy Kepner. Big data dimensional analysis. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[24] Vijay Gadepally and Jeremy Kepner. Using a power law distribution to describe big data. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2015.

[25] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G Mattson. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3211–3220. IEEE, 2017.

[26] Michael Hausenblas and Jacques Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big data*, 1(2):100–104, 2013.

[27] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The bigdawg polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[28] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884, 2014.

[29] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. Demonstrating the bigdawg polystore system for ocean metagenomics analysis. In *CIDR*, 2017.

[30] Aaron J Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment*, 8(12):1908, 2015.

[31] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. *PVLDB*, 11(11):1454–1467, 2018.

[32] Dong Deng, Wenbo Tao, Ziawasch Abedjan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Guoliang Li, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Unsupervised string transformation learning

for entity consolidation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 196–207, 2019.

[33] mlflow. https://mlflow.org. Accessed: March 2019.

[34] Apache Airflow. https://airflow.apache.org. Accessed: March 2019.

[35] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya G. Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *PVLDB*, 2018.

[36] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. The return of jedai: End-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

[37] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeffrey F. Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[38] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[39] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, 2017.

[40] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 8(11):1310–1321, 2015.

[41] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd D. Millstein, and Miryung Kim. Bigdebug: debugging primitives for interactive big data processing in spark. In *ICSE*, pages 784–795. ACM, 2016.

[42] Essam Mansour, Dong Deng, Raul Castro Fernandez, Abdulhakim Ali Qahtan, Wenbo Tao, Ziawasch Abedjan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Building data civilizer pipelines with an advanced workflow engine. In *ICDE*, 2018.

[43] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: a system for machine learning model management. In *HILDA*, 2016.

[44] Wenbo Tao, Xiaoyu Liu, Çagatay Demiralp, Remco Chang, and Michael Stonebraker. Kyrix: Interactive visual data exploration at scale. In *CIDR*, 2019.

[45] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, September 2014.

[46] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, pages 722–735, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] Amit Singhal. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html, May 2012. [Online; accessed May 30, 2019].

[48] TheUniProtConsortium. Uniprot: a worldwide hub of protein knowledge. In *Nucleic Acids Research*, 2018.

[49] Welcome to musicbrainz! https://musicbrainz.org/, 2019. [Online; accessed May 30, 2019].

[50] Geonames. http://www.geonames.org/, 2019. [Online; accessed May 30, 2019].

[51] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[52] Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in knowitall: (preliminary results). In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 100–110, 2004.

[53] Christian Bizer. The emerging web of linked data. *IEEE Intelligent Systems*, 24(5):87–92, September 2009.

[54] Derry Tanti Wijaya and Tom M. Mitchell. Mapping verbs in different languages to knowledge base relations using web text as interlingua. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 818–827, 2016.

[55] Rahul Gupta, Alon Y. Halevy, Xuezhi Wang, Steven Euijong Whang, and Fei Wu. Biperpedia: An ontology for search applications. *PVLDB*, 7(7):505–516, 2014.

[56] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392, 2016.

[57] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *CoRR*, abs/1806.03822, 2018.

[58] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *CoRR*, abs/1611.01603, 2016.

[59] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218, 2012.

[60] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. Amazon neptune: Graph data management in the cloud. In *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018.

[61] Wolfgang Gatterbauer, Paul Bohunsky, Marcus Herzog, Bernhard Krüpl, and Bernhard Pollak. Towards domain-independent information extraction from web tables. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 71–80, 2007.

[62] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. Snorkel: A system for lightweight extraction. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[63] Michael J. Cafarella, Alon Y. Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. Uncovering the relational web. In *11th International Workshop on the Web and Databases, WebDB 2008, Vancouver, BC, Canada, June 13, 2008*, 2008.

[64] Alexander Yates and Oren Etzioni. Unsupervised methods for determining object and relation synonyms on the web. *CoRR*, abs/1401.5696, 2014.

[65] Sreeram Balakrishnan, Alon Y. Halevy, Boulos Harb, Hongrae Lee, Jayant Madhavan, Afshin Rostamizadeh, Warren Shen, Kenneth Wilder, Fei Wu,

and Cong Yu. Applying webtables in practice. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[66] Ari Kobren, Thomas L. Logan, Siddarth Sampangi, and Andrew J. Mccallum. Domain specific knowledge base construction via crowdsourcing. 2014.

[67] Ari Kobren, Nicholas Monath, and Andrew McCallum. Entity-centric attribute feedback for interactive knowledge bases. 2017.

[68] Gabor Angeli, Julie Tibshirani, Jean Wu, and Christopher D. Manning. Combining distant and partial supervision for relation extraction. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1556–1567, Doha, Qatar, October 2014. Association for Computational Linguistics.

[69] Angli Liu, Stephen Soderland, Jonathan Bragg, Christopher H. Lin, Xiao Ling, and Daniel S. Weld. Effective crowd annotation for relation extraction. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 897–906, San Diego, California, June 2016. Association for Computational Linguistics.

[70] Anthony Fader, Luke Zettlemoyer, and Oren Etzioni. Open question answering over curated and extracted knowledge bases. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1156–1165, 2014.

[71] NationalResearchCouncil. Toward precision medicine: Building a knowledge network for biomedical research and a new taxonomy of disease. From the National Research Council Committee on A Framework for Developing a New Taxonomy of Disease, 01 2012.

[72] Ruobing Xie, Zhiyuan Liu, and Maosong Sun. Representation learning of knowledge graphs with hierarchical types. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2965–2971. IJCAI/AAAI Press, 2016.

[73] Yanjie Wang, Daniel Ruffinelli, Rainer Gemulla, Samuel Broscheit, and Christian Meilicke. On evaluating embedding models for knowledge base completion. In Isabelle Augenstein, Spandana Gella, Sebastian Ruder, Katharina Kann, Burcu Can, Johannes Welbl, Alexis Conneau, Xiang Ren, and Marek Rei, editors, *Proceedings of the 4th Workshop on Representation Learning for NLP, RepL4NLP@ACL 2019, Florence, Italy, August 2, 2019*, pages 104–112. Association for Computational Linguistics, 2019.

[74] Zhongjun Jin, Christopher Baik, Michael J. Cafarella, and H. V. Jagadish. Beaver: Towards a declarative schema mapping. In Carsten Binnig, Juliana Freire, and Eugene Wu, editors, *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 10:1–10:4. ACM, 2018.

[75] Zhongjun Jin, Christopher Baik, Michael J. Cafarella, H. V. Jagadish, and Yuze Lou. Demonstration of a multiresolution schema mapping system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[76] Li Qian, Michael J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 73–84. ACM, 2012.

[77] El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael Stonebraker. Dagger: A data (not code) debugger. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[78] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26, 10 2017.

[79] Holger Stitz, S. Luger, Marc Streit, and N. Gehlenborg. Avocado: Visualization of workflow-derived data provenance for reproducible biomedical research. *Computer Graphics Forum*, 35:481–490, 06 2016.

[80] Louis Bavoil, Steven Callahan, Carlos Scheidegger, Patricia Crossno, Claudio Silva, and Juliana Freire. Vistrails: Enabling interactive multiple-view visualizations. page 18, 01 2005.

[81] Chenyun Dai, Dan Lin, Elisa Bertino, and Murat Kantarcioglu. An approach to evaluate data trustworthiness based on data provenance. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, pages 82–98, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[82] Olaf Hartig and Jun Zhao. Using web data provenance for quality assessment. In *Proceedings of the First International Conference on Semantic Web in Provenance Management - Volume 526*, SWPM'09, page 29–34, Aachen, DEU, 2009. CEUR-WS.org.

[83] Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. Orchestra: Facilitating collaborative data sharing. In *Proceedings of the 2007 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '07, page 1131–1133, New York, NY, USA, 2007. Association for Computing Machinery.

[84] Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, Ken Goldberg, and Tim Kraska. Privateclean: Data cleaning and differential privacy. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 937–951, New York, NY, USA, 2016. Association for Computing Machinery.

[85] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.

[86] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 683–698, 2017.

[87] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. Foofah: A programming-by-example system for synthesizing data transformation programs. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1607–1610, 2017.

[88] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 458–469, 2013.

[89] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. Sampling from repairs of conditional functional dependency violations. *The VLDB Journal*, 23(1):103–128, February 2014.

[90] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*, pages 1285–1300, 2018.

[91] El Kindi Rezig, Mourad Ouzzani, Ahmed K. Elmagarmid, Walid G. Aref, and Michael Stonebraker. Towards an end-to-end human-centric data cleaning framework. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 1:1–1:7. ACM, 2019.

[92] Vijay Gadepally, Braden Hancock, Benjamin Kaiser, Jeremy Kepner, Pete Michaleas, Mayank Varia, and Arkady Yerukhimovich. Computing on masked data to improve the security of big data. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6. IEEE, 2015.

[93] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[94] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system.

[95] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE, 2014.

[96] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191. IEEE, 2017.

[97] Richard Shay, Uri Blumenthal, Vijay Gadepally, Ariel Hamlin, John Darby Mitchell, and Robert K Cunningham. Don't even ask: Database access control through query control. *ACM SIGMOD Record*, 47(3):17–22, 2019.

[98] Lori Aratani. Secret use of census info helped send japanese americans to internment camps in wwii. *The Washington Post*, Apr 2018.

[99] Jeremy Kepner, Vijay Gadepally, Pete Michaleas, Nabil Schear, Mayank Varia, Arkady Yerukhimovich, and Robert K Cunningham. Computing on masked data: a high performance method for improving big data veracity. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[100] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[101] Protecting Aggregated Data. Technical report, US-CERT, 12 2005.

[102] Vijay Gadepally, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Lauren Edwards, Matthew Hubbell, Peter Michaleas, Julie Mullen, et al. D4m: Bringing associative arrays to database engines. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2015.

[103] Andrew Pavlo and Matthew Aslett. What's really new with newsql? *ACM Sigmod Record*, 45(2):45–55, 2016.

[104] Ankush M Gupta, Vijay Gadepally, and Michael Stonebraker. Cross-engine query execution in federated database systems. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[105] Peinan Chen, Vijay Gadepally, and Michael Stonebraker. The bigdawg monitoring framework. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.