Toward a System Building Agenda for Data Integration (and Data Science)

AnHai Doan, Pradap Konda, Paul Suganthan G.C., Adel Ardalan, Jeffrey R. Ballard, Sanjib Das,
Yash Govind, Han Li, Philip Martinkus, Sidharth Mudgal, Erik Paulson, Haojun Zhang
University of Wisconsin-Madison

Abstract

We argue that the data integration (DI) community should devote far more effort to building systems, in order to truly advance the field. We discuss the limitations of current DI systems, and point out that there is already an existing popular DI "system" out there, which is PyData, the open-source ecosystem of 138,000+ interoperable Python packages. We argue that rather than building isolated monolithic DI systems, we should consider extending this PyData "system", by developing more Python packages that solve DI problems for the users of PyData. We discuss how extending PyData enables us to pursue an integrated agenda of research, system development, education, and outreach in DI, which in turn can position our community to become a key player in data science. Finally, we discuss ongoing work at Wisconsin, which suggests that this agenda is highly promising and raises many interesting challenges.

1 Introduction

In this paper we focus on data integration (DI), broadly interpreted as covering all major data preparation steps such as data extraction, exploration, profiling, cleaning, matching, and merging [10]. This topic is also known as data wrangling, munging, curation, unification, fusion, preparation, and more. Over the past few decades, DI has received much attention (e.g., [37, 29, 31, 20, 34, 33, 6, 17, 39, 22, 23, 5, 8, 36, 15, 35, 4, 25, 38, 26, 32, 19, 2, 12, 5, 16, 2, 3]). Today, as data science grows, DI is receiving even more attention. This is because many data science applications must first perform DI to combine the raw data from multiple sources, before analysis can be carried out to extract insights.

Yet despite all this attention, today we do not really know whether the field is making good progress. The vast majority of DI works (with the exception of efforts such as Tamr and Trifacta [36, 15]) have focused on developing *algorithmic solutions*. But we know very little about whether these (ever-more-complex) algorithms are indeed useful in practice. The field has also built mostly isolated *system prototypes*, which are hard to use and combine, and are often not powerful enough for real-world applications. This makes it difficult to decide what to *teach* in DI classes. Teaching complex DI algorithms and asking students to do projects using our prototype systems can train them well for doing DI research, but are not likely to train them well for solving real-world DI problems in later jobs. Similarly, *outreach to real users* (e.g., domain scientists) is difficult. Given that we have mostly focused on "point DI problems", we do not know how to help them solve *end-to-end* DI tasks. That is,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

we cannot tell them how to start, what algorithms to consider, what systems to use, and what they need to do manually in each step of the DI process.

In short, today our DI effort in research, system development, education, and outreach seem disjointed from one another, and disconnected from real-world applications. As data science grows, this state of affairs makes it hard to figure out how we can best relate and contribute to this major new field.

In this paper we take the first steps in addressing these problems. We begin by arguing that the key to move forward (and indeed, to tie everything together) is to devote far more effort to *building DI systems*. DI is engineering by nature. We cannot just keep developing DI algorithmic solutions in a vacuum. At some point we need to build systems and work with real users to evaluate these algorithms, to integrate disparate R&D efforts, and to make practical impacts. In this aspect, DI can take inspiration from RDBMSs and Big Data systems. Pioneering systems such as System R, Ingres, Hadoop, and Spark have really helped push these fields forward, by helping to evaluate research ideas, providing an architectural blueprint for the entire community to focus on, facilitating more advanced systems, and making widespread real-world impacts.

We then discuss the limitations of current DI systems, and point out that there is already an existing DI system out there, which is very popular and growing rapidly. This "system" is PyData, the open-source ecosystem of 138,000+ interoperable Python packages such as pandas, matplotlib, scikit-learn, etc. We argue that rather than building isolated monolithic DI systems, *the DI community should consider extending this PyData "system", by developing Python packages that can interoperate and be easily combined to solve DI problems for the users of PyData*. This can address the limitations of the current DI systems, provide a system for the entire DI community to rally around, and in general bring numerous benefits and maximize our impacts.

We propose to extend the above PyData "system" in three ways:

- For each end-to-end DI scenario (e.g., entity matching with a desired F_1 accuracy), develop a "how-to guide" that tells a power user (i.e., someone who can program) how to execute the DI process step by step, identify the true "pain points" in this process, develop algorithmic solutions for these pain points, then implement the solutions as Python packages.

- Foster PyDI, an ecosystem of such DI packages as a part of PyData, focusing on how to incentivize the community to grow PyDI, how to make these packages seamlessly interoperate, and how to combine them to solve larger DI problems.

- Extend PyDI to the cloud, collaborative (including crowdsourcing), and lay user settings.

We discuss how extending PyData can enable our community to pursue an *integrated* agenda of research, system development, education, and outreach. In this agenda we develop solutions for real-world problems that arise from solving end-to-end DI scenarios, build real-world tools into PyData, then work with students and real-world users on using these tools to solve DI problems. We discuss how this agenda can position our community to become a key player in data science, who "owns" the data quality part of this new field. Finally, we describe initial work on this agenda in the past four years at Wisconsin. Our experience suggests that this agenda is highly promising and raises numerous interesting challenges in research, systems, education, and outreach.

2 Data Science and Data Integration

In this section we briefly discuss data science, data integration, and the relationship between the two.

Currently there is no consensus definition for data science (DS). For our purposes, we will define data science as a field that develops principles, algorithms, tools, and best practices to manage data, focusing on three topics: (a) analyzing raw data to infer insights, (b) building data-intensive artifacts (e.g., recommender systems, knowledge bases), and (c) designing data-intensive experiments to answer questions (e.g., A/B testing). As such, DS is clearly here to stay (even though the name may change), for the simple reason that everything is now data driven, and will only become even more so in the future.

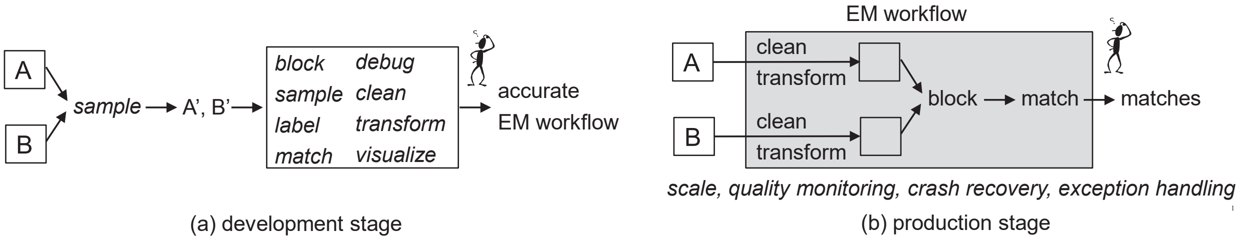

Figure 1: Matching two tables in practice often involves two stages and many steps (shown in italics).

In this paper we focus on the first topic, analyzing raw data to infer insights, which has received a lot of attention. A DS task here typically consists of two stages. In the *data integration (DI) stage*, raw data from many different sources is acquired and combined into a single clean integrated dataset. Then in the *data analysis stage*, analysis is performed on this dataset to obtain insights. Both stages extensively use techniques such as visualization, learning, crowdsourcing, Big Data scaling, and statistics, among others.

Core DI Problems & End-to-End DI Scenarios: The DI stage is also known as data wrangling, preparation, curation, cleaning, munging, etc. Major problems of this stage include extraction, exploration, profiling, cleaning, transforming, schema matching, entity matching, merging, etc. We refer to these as *core DI problems*. When solving a core DI problem, real-world users often want to reach a desired outcome (e.g., at least 95% precision and 80% recall). We refer to such desired outcomes as *goals*, and to such scenarios, which go from the raw data to a goal, as *end-to-end DI scenarios*. (As a counter example, simply trying to maximize the F_1 accuracy of entity matching, as many current research works do, is not an end-to-end scenario.)

Development and Production Stages: To solve an end-to-end DI scenario, a user typically goes through two stages. In the *development stage*, the user experiments to find a DI workflow that can go from raw data to the desired goal. This is often done using data samples. In this stage, the user often has to explore to understand the problem definition, data, and tools, and make changes to them if necessary. Then in the *production stage*, the user specifies the discovered workflow (e.g., declaratively or using a GUI), optionally optimizes, then executes the workflow on the entirety of data. (Sometimes the steps of these two stages may be interleaved.)

Example 1: *Consider matching two tables A and B each having 1M tuples, i.e., find all pairs $(a \in A, b \in B)$ that refer to the same real-world entity. In the development stage (Figure 1.a), user U tries to find an accurate EM workflow. This is often done using data samples. Specifically, U first samples two smaller tables A' and B' (each having 100K tuples, say) from A and B. Next, U performs blocking on A' and B' to remove obviously non-matched tuple pairs. U often must try and debug different blocking techniques to find the best one.*

Suppose U wants to apply supervised learning to match the tuple pairs that survive the blocking step. Then next, U may take a sample S from the set of such pairs, label pairs in S (as matched / non-matched), and then use the labeled sample to develop a learning-based matcher (e.g., a classifier). U often must try and debug different learning techniques to develop the best matcher. Once U is satisfied with the accuracy of the matcher, the production stage begins (Figure 1.b). In this stage, U executes the EM workflow that consists of the developed blocking strategy followed by the matcher on the original tables A and B. To scale, U may need to rewrite the code for blocking and matching to use Hadoop or Spark.

3 Limitations of Current Data Integration Systems

Each current DI system tries to solve either *a single core DI problem* or *multiple core DI problems jointly* (e.g., schema matching, followed by schema integration, then EM). We now discuss these two groups in turn. Consider systems for a single core DI problem. Our experience suggests that this group suffers from the following limitations.

1. Do Not Solve All Stages of the End-to-End DI Process: Most current DI systems support only the production stage. For example, most current EM systems provide a set of blockers and matchers. The user can specify an EM workflow using these blockers/matchers (either declaratively or via a GUI). The systems then optimize and execute the EM workflow. Much effort has been devoted to developing effective blocker/matcher operators (e.g., maximizing accuracy, minimizing runtime, minimizing crowdsourcing cost, etc.). There has been relatively little work on the development stage. It is possible that database researchers have focused mostly on the production stage because it follows the familiar query processing paradigm of RDBMSs. Regardless, we cannot build practical DI systems unless we also solve the development stage.

2. Provide No How-To Guides for Users: Solving the development stage is highly non-trivial. There are three main approaches. First, we can try to completely automate it. This is unrealistic. Second, we can still try to automate, but allowing limited human feedback at various points. This approach is also unlikely to work. The main reason is that the development stage is often very messy, requiring multiple iterations involving many subjective judgments from the human user. Very often, after working in this stage for a while, the user gains a better understanding of the problem and the data at hand, then revises many decisions on the fly.

Example 2: *Consider the labeling step in Example 1. Labeling tuple pairs in sample S as matched or non-matched seems trivial. Yet it is actually quite complicated in practice. Very often, during the labeling process user U gradually realizes that his/her current match definition is incorrect or inadequate. For instance, when matching restaurant descriptions, U may start with the definition that two restaurants match if their names and street addresses match. But after a while, U realizes that the data contains many restaurants that are branches of the same chain (e.g., KFC). After checking with the business team, U decides that these should match too, even though their street addresses do not match. Revising the match definition however requires U to revisit and potentially relabel pairs that have already been labeled, a tedious and time-consuming process.*

As another example, suppose a user U wants to perform EM with at least 95% precision and 80% recall. How should U start? Should U use a learning-based or a rule-based EM approach? What should U do if after many tries U still cannot reach 80% recall with a learning-based approach? It is unlikely that an automated approach with limited human feedback would work for this scenario.

As a result, it is difficult to imagine that the development stage can be automated with any reasonable degree soon. In fact, today it is still often executed using the third approach, where a human user drives the end-to-end process, making decisions and using (semi-)automated tools in an *ad-hoc* fashion. Given this situation, many users have indicated to us that what they really need, first and foremost, is *a how-to guide* on how to execute the development stage. Such a guide is *not* a user manual on how to use a tool. Rather, it is a detailed step-by-step instruction to the user on how to start, when to use which tools, and when to do what manually. Put differently, *it is an (often complex) algorithm for the human user to follow*. Current DI systems lack such how-to guides.

3. Provide Few Tools for the Pain Points: When executing the development state, a user often runs into many "pain points" and wants (semi-)automated tools to solve them. But current DI systems have provided few such tools. Some pain points are well known, e.g., debugging blockers/matchers in EM. Many more are not well known today. For example, many issues thought trivial turn out to be major pain points in practice.

Example 3: *Exploring a large table by browsing around is a major pain point, for which there is no effective tool today (most users still use Excel, OpenRefine, or some limited browsing capabilities in PyData). Counting the missing values of a column, which seems trivial, turns out to be another major pain point. This is because in practice, missing values are often indicated by a wide range of strings, e.g., "null", "none", "N/A", "unk", "unknown", "-1", "999", etc. So the user often must painstakingly detect and normalize all these synonyms, before being able to count. Labeling tuple pairs as match/no-match in EM is another major pain point. Before labeling, users often want to run a tool that processes the tuple pairs and highlights possible match definitions, so that they can develop the most comprehensive match definition. Then during the labeling process, if users must still revise the match definition, they want a tool that quickly flags already-labeled pairs that may need to be relabeled.*

4. Difficult to Exploit a Wide Variety of Capabilities: It turns out that even when we just try to solve a single core DI problem, we already have to utilize a wide variety of capabilities. For example, when doing EM, we often have to utilize SQL querying, keyword search, learning, visualization, crowdsourcing, etc. Interestingly, we also often have to solve *other DI problems*, such as exploration, cleaning, information extraction, etc. So we need the solution capabilities for those problems as well.

Today, it is very difficult to exploit all these capabilities. Incorporating all of them into a single DI system is difficult, if not impossible. The alternative solution of moving data among multiple systems, e.g., an EM system, an extraction system, a visualization system, etc., also does not work. This is because solving a DI problem is often an iterative process. So we would end up moving data among multiple systems *repeatedly*, often by reading/writing to disk and translating among proprietary data formats numerous times, in a tedious and time consuming process. A fundamental problem here is that most current DI systems are *stand-alone monoliths* that are not designed to interoperate with other systems. Put differently, most current DI researchers are still building stand-alone systems, rather than developing an ecosystem of interoperable DI systems.

5. Difficult to Customize, Extend, and Patch: In practice, users often want to customize a generic DI system to a particular domain. Users also often want to extend the system with latest technical advances, e.g., crowdsourcing, deep learning. Finally, users often have to write code, e.g., to implement a lacking functionality or combine system components. Writing "patching" code correctly in "one shot" (i.e., one iteration) is difficult. Hence, ideally such coding should be done in an interactive scripting environment, to enable rapid prototyping and iteration. Few if any of the current DI systems are designed from scratch such that users can easily customize, extend, and patch in many flexible ways. Most systems provide "hooks" at only certain points in the DI pipeline for adding limited new functionalities (e.g., a new blocker/matcher), and the vast majority of systems are not situated in an interactive scripting environment, making patching difficult.

6. Similar Problems for the Production Stage: So far we have mostly discussed problems with the development stage. But it appears that many of these problems may show up in the production stage too. Consider for example a domain scientist U trying to execute an EM workflow in the production stage on a single desktop machine and it runs too slowly. What should U do next? Should U try a machine with bigger memory or disk? Should U try to make sure the code indeed runs on multiple cores? Should U try some of the latest scaling techniques such as Dask ($dask.pydata.org$), or switch to Hadoop or Spark? Today there is no guidance to such users on how best to scale a DI workflow in production.

Limitations of Systems for Multiple DI Problems: So far we have discussed systems for a single core DI problem. We now discuss systems for multiple DI problems. Such a system jointly solves a set of DI problems, e.g., data cleaning, schema matching and integration, then EM. This helps users solve the DI application seamlessly end-to-end (without having to switch among multiple systems), and enables runtime/accuracy optimization across tasks. Our experience suggests that these systems suffer from the following limitations. (1) For each component DI problem, these systems have the same problems as the systems for a single DI problems. (2) As should be clear by now, building a system to solve a single core DI problem is already very complex. Trying to solve multiple such problems (and accounting for the interactions among them) in the same system often exponentially magnifies the complexity. (3) To manage this complexity, the solution for each component problem is often "watered down", e.g., fewer tools are provided for both the development and production stages. This in turn makes the system less useful in practice. (4) If users want to solve just 1-2 DI problems, they still need to install and load the entire system, a cumbersome process. (5) In many cases optimization across problems (during production) does not work, because users want to execute the problems one by one and materialize their outputs on disk for quality monitoring and crash recovery. (6) Finally, such systems often handle only a pre-specified set of workflows that involves DI problems from a pre-specified set. If users want to try a different workflow or need to handle an extra DI problem, they need another system, and so end up combining multiple DI systems anyway.

4 The PyData Ecosystem of Open-Source Data Science Tools

In the past decade, using open-source tools to do data science (for both data integration and data analysis stages) has received significant growing attention. The two most well-known ecosystems of such open-source tools are in Python and R. In this paper we focus on the Python ecosystem, popularly known as PyData.

What Do They Do? First and foremost, the PyData community has been building a variety of tools (typically released as Python packages). These tools seek to solve data problems (e.g., Web crawling, data acquisition, extraction), implement cross-cutting techniques (e.g., learning, visualization), and help users manage their work (e.g., Jupyter notebook). As of May 2018, there are 138,000+ packages available on *pypi.org* (compared to "just" 86,000 packages in August 2016). Popular packages include NumPy (49M downloads), pandas (27.7M downloads), matplotlib (13.8M downloads), scikit-learn (20.9M downloads), jupyter (4.9M downloads), etc.

The community has also developed extensive software infrastructure to build tools, and ways to manage/package/distribute tools. Examples include nose, setuptools, *pypi.org*, anaconda, conda-forge, etc. They have also been extensively educating developers and users, using books, tutorials, conferences, etc. Recent conferences include PyData (with many conferences per year, see *pydata.org*), JupyterCon, AnacondaCon, and more. Universities also often hold many annual Data Carpentry Workshops (see *datacarpentry.org*) to train students and scientists in working with PyData.

Finally, the PyData community has fostered many players (companies, non-profits, groups at universities) to work on the above issues. Examples include Anaconda Inc (formerly Continuum Analytics, which releases the popular anaconda distribution of selected PyData packages), NumFocus (a non-profit organization that supports many PyData projects), *datacarpentry.org* (building communities teaching universal data literacy), *software-carpentry.org* (teaching basic lab skills for research computing), and more.

Why Are They Successful? Our experience suggests four main reasons. The first obvious reason is that PyData tools are free and open-source, making it cheap and easy for a wide variety of users to use and customize. Many domain scientists in particular prefer open-source tools, because they are free, easy to install and use, and can better ensure transparency and reproducibility (than "blackbox" commercial software). The second reason, also somewhat obvious, is the extensive community effort to assist developers and users, as detailed earlier. The third, less obvious, reason is that PyData tools are *practical*, i.e., they often are developed to address *creators' pain points*. Other users doing the same task often have the same pain points and thus find these tools useful.

Finally, the most important reason, in our opinion, is *the conscious and extensive effort to develop an ecosystem of interoperable tools and the ease of interoperability of these tools*. As discussed earlier, solving DI problems often requires many capabilities (e.g., exploration, visualization, etc.). No single tool today can offer all such capabilities, so DI (and DS) is often done by using a set of tools, each offering some capabilities. For this to work, *tool interoperability is critical*, and PyData appears to do this far better than any other DI platforms, in the following ways. (a) The interactive Python environment makes it easy to interoperate: one just has to import a new tool and the tool can immediately work on existing data structures already in memory. (b) Tool creators understand the importance of interoperability and thus often consciously try to make tools easy to interoperate. (c) Much community effort has also been spent on making popular tools easy to interoperate. For example, for many years Anaconda Inc. has been selecting the most popular PyData packages (536 as of May 2018), curating and making sure that they can interoperate well, then releasing them as the popular anaconda data science platform (with over 4.5M users, as of May 2018).

What Are Their Problems? From DI perspectives we observe several problems. First, even though PyData packages cover all major steps of DI, they are very weak in certain steps. For example, there are many outstanding packages for data acquisition, exploration (e.g., using visualization and statistics), and transformation. But until recently there are few good packages for string matching and similarity join, schema matching, and entity matching, among others. Second, there is very little guidance on how to solve DI problems. For example, there is very little published discussion on the challenges and solutions for missing values, string matching, entity/schema matching, etc. Third, most current PyData packages do not scale to data larger than memory and to multicore/machine cluster settings (though solutions such as Dask have been proposed). Finally, building data tools that interoperate raises many challenges, e.g., how to manage metadata/missing values/type mismatch across packages. Currently only some of these challenges have been addressed, in an ad-hoc fashion. Clearly, solving all of these problems can significantly benefit from the deep expertise of the DI research community.

5 The Proposed System Building Agenda

Based on the extensive discussion in the previous section, we propose that our community consider extending the PyData "system" to solve DI problems for its users. First, this "system" already exists, with millions of users. So it is important that we consider helping them. (A similar argument was made for XML, MapReduce, etc: regardless of what one thinks about these, many users are using them and we ought to help these users if we can.) Second, we believe that the system building template of PyData is worth exploring. As mentioned earlier, no single uber-system can provide all DI capabilities. An ecosystem of interoperable DI tools situated in an interactive environment (e.g., Python) seems highly promising.

Third, extending PyData brings numerous practical benefits and helps maximize our impacts. (a) By building on PyData, we can instantly leverage many capabilities, thus avoid building weak research prototypes. (b) Many domain scientists use this "system". So by extending it, we can better convince them to use our tools. (c) We can teach PyData and our tools seamlessly in classes, thereby educating students in practical DS tools (that they should know for later jobs) yet obtaining an evaluation of our tools. (d) For our students, developing Python packages are much easier compared to developing complex stand-alone systems (such as RDBMSs). So academic researchers stand a better chance of developing "systems components" that are used in practice. (e) Tools developed by different DI groups have a better chance of being able to work together. (f) PyData is weak in many DI capabilities. So this is a real opportunity for our community to contribute. (g) Finally, if the whole community focuses on a single system, we have a better chance of measuring progress.

Specifically, we propose the following concrete agenda:

- Build systems, each of which helps power users solve an end-to-end scenario involving a single core DI problem, as software packages in PyData.
- Foster PyDI, an ecosystem of such DI software packages as a part of PyData, focusing on how to incentivize the community to grow PyDI, how to make these packages seamlessly interoperate, and how to combine them to solve larger DI problems.
- Extend PyDI to the cloud, collaborative (including crowdsourcing), and lay user settings.

We now motivate and discuss these directions.

5.1 Build Systems for Core DI Problems

To build these systems, we propose the following steps (see Figure 2 for the proposed architecture).

1. Identify a Concrete End-to-End DI Scenario: We propose to start by identifying a concrete scenario that involves *a single core DI problem*. (Later, as we know much better how to solve core DI problems, we can leverage that knowledge to build "composite systems" to jointly solve multiple core DI problems.) This concrete scenario must be *end-to-end (e2e)*, i.e., going from raw data to a desired outcome for the end user (e.g., EM with at least 95% precision and 90% recall). Our goal is to solve this end-to-end scenario for *power users*: those who may not be DI experts but can code, e.g., data scientists. (Later we can leverage these solutions to develop solutions for *lay users*, i.e., those that cannot code, in a way analogous to building on assembly languages to develop higher-level CS languages.)

2. Develop a Detailed How-To Guide: Next, we should develop a detailed how-to guide to the user on how to execute the above e2e scenario, step by step. First, the guide should tell the user whether he or she should execute the scenario in stages. As discussed earlier, there are typically two stages to consider: development and production. Depending on the scenario, these stages may be executed separately or interleaved somehow. For example, if the user wants to match two tables of 1M tuples each, then the best course of action is to do a development stage first with data samples to find an accurate EM workflow, then execute this workflow on the two original tables in a production stage. On the other hand, if the two tables have just 300 tuples each, then the user can execute the two stages interleavingly "in one shot".

Then for each stage, the guide should tell the user as clearly as possible how it should be executed, step by step: which step should be first, second, etc. For each step, the guide in turn should provide as detailed instruction as possible on how to execute it. In short, the guide is a detailed *algorithms* for the human user. A "rule of thumb" is that if the user knows how to code, he or she should be able to use the guide to execute the e2e scenario, *even without utilizing any tool* (of course, this can take a long time, but the key is that the user should be able to do it). In practice, the guide can utilize any appropriate (semi-)automatic existing tools.

3. Identify Pain Points in the How-To Guide: Next, we should examine the guide carefully to identify real pain points, i.e., steps that are laborious or time consuming for the user and can be fully or partly automated. These include well-known pain points (e.g., performing blocking in EM) or steps that are thought trivial but turn out to be major problems in practice (e.g., browsing a large table, managing missing values, labeling, see Example 3) .

4. Develop Solutions and Tools for the Pain Points: Finally, we should develop algorithmic (semi-)automated solutions for the pain points, then implement them as tools. Each tool is a set of Python packages, often using other packages in PyData. For example, tools for the pain points of the development stage can use packages in the Python Data Analysis Stack (e.g., pandas, scikit-learn, numpy, etc.), while tools for the production stage, where scaling is a major focus, can use packages in the Python Big Data stack to perform MapReduce (e.g., Pydoop, mrjob), Spark (e.g., PySpark), and parallel/distributed computing in general (e.g., pp, dispy).

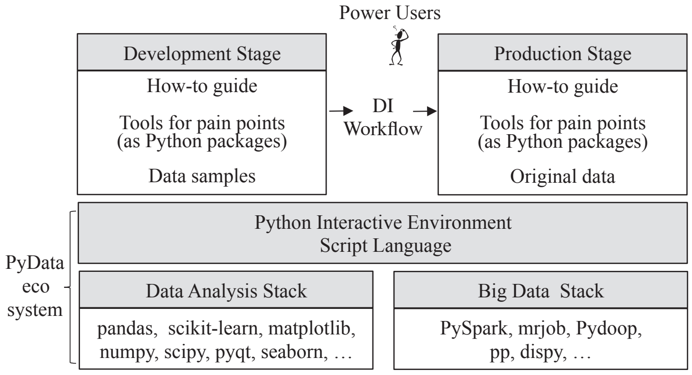
Power Users
Development Stage
How-to guide
Tools for pain points (as Python packages)
Data samples
DI Workflow
Production Stage
How-to guide
Tools for pain points (as Python packages)
Original data
Python Interactive Environment
Script Language
PyData eco system
Data Analysis Stack
pandas, scikit-learn, matplotlib, numpy, scipy, pyqt, seaborn, …
Big Data Stack
PySpark, mrjob, Pydoop, pp, dispy, …

Figure 2: Proposed architecture for DI systems.

Example 4: *We have built* Magellan*, a system that helps users solve the EM problem scenario described in Example 1 [17]. We provide detailed how-to guides, then develop tools for the pain points. For the development stage, these tools (a) take a sample from two tables A and B (ensuring a reasonable number of matches in the sample), (b) debug the blockers, (c) debug the labeling process, (d) select the best matcher, and (e) debug the matchers, among others [24, 27, 28]. There are 104 Python commands that users can use. By leveraging 11 packages in the PyData ecosystem, we were able to develop these tools with relatively little effort. For the production stage, we have developed a preliminary how-to guide, which tells the user how to scale. But more work is necessary to refine the guide.*

We have also been building DI systems to normalize attribute values, count the number of missing values in a column, taking into account synonyms for missing values, and help users clean a table, using a detailed how-to guide which tells him or her which attribute to clean in which order, and how.

5.2 Foster a DI Ecosystem as a Part of PyData

In the second part of our system building agenda, we propose to foster PyDI, an ecosystem of interoperable DI software packages as a part of PyData. To do so, there are several obvious actions we should take. (1) PyData has been very successful. We should study it and similar ecosystems (e.g., R, Bioconductor), then apply the lessons learned to grow PyDI. (2) We should solve a wide variety of end-to-end DI problem scenarios and build many more tools as Python packages, as discussed earlier. (3) We should work extensively on helping PyDI researchers, developers, educators and users, in terms of infrastructure, books, conferences, tutorials, community resources, etc. To recruit more developers and users, we can apply the lessons learned from PyData/R, "piggyback" on their efforts (e.g., promoting our PyDI tools in Data Carpentry Workshops), and recruit data science students and domain scientists as our users.

Must Ensure That Tools Can Interoperate: Critical to our success of fostering PyDI, however, is the issue of tool interoperability. As described earlier, the proposed DI systems (i.e., tools) will be in the PyData ecosystem and expected to "play well" with other packages. We say that these systems are "open-world", in contrast to current stand-alone "closed-world" DI systems. It is critical that we design these "open-world" DI systems from scratch for interoperability, so that they can easily exploit the full power of PyData and can be seamlessly combined to solve multiple DI problems.

In particular, these systems should expect other systems (in the ecosystem) to be able to manipulate their own data, they may also be called upon by other systems to manipulate those systems' data, and they should be designed in a way that facilitates such interaction. This raises many interesting technical challenges. For example, what kinds of data structures should a tool T use to facilitate interoperability? How to manage metadata if any external tool can modify the data of T, potentially invalidating T's metadata without T knowing about it? How to manage missing values, data type mismatches, version incompatibilities, etc. across the packages? See [17] for a discussion of some of these issues. We should identify and develop solutions for these challenges.

Finally, the PyData experience suggests that it might not be sufficient to just rely on individual creators to make the tools interoperate. Community players may have considerably more resources for cleaning/combining packages and making them work together well. So it is important that we foster such players (e.g., startups, non-profits, research labs, data science institutes, etc.).

5.3 Build Cloud/Collaborative/Lay User Versions of DI Systems

So far we have proposed to develop DI systems for a single power user, in his/her local environment. There are however increasingly many more DI settings, which can be characterized by *people* (e.g., power user, lay user, a team of users, etc.) and *technologies* (e.g., cloud, crowdsourcing, etc.). For instance, a team of scientists wants to solve a DI problem collaboratively, or a lay user wants to do cloud-based DI because he/she does not know how to run a local cluster. To maximize the impacts of our DI systems, we should also consider these settings. In particular, we briefly discuss three important settings below, and show that these settings can build on DI systems proposed so far, but raise many additional R&D challenges.

Cloud-Based DI Systems: Many recent efforts to push PyData "into the cloud" have developed *infrastructure as a service (IaaS)*. Such a service allows a scientist to quickly create a *cloud environment for data science*, by renting machines then installing a wide variety of data science software, including PyData packages (many of which may already been pre-installed). Scientists can then collaborate in this cloud environment. There have been far fewer efforts, however, to develop *software as a service (SaaS)* and *platform as a service (PaaS)* to solve DI problems. We should develop such services. If we can develop cloud services to perform DI tasks (e.g., profiling, cleaning, labeling, matching, etc.), that would bring major benefits. For example, scientists can easily collaborate (e.g., in labeling a dataset), share data (e.g., among team members in multiple locations), remotely monitor task progress, elastically rent more machines/memory to scale, and get access to resources not available locally (e.g., GPU), among others. Further, if we can develop cloud *platform* services for DI, they can save developers a lot of work when building new cloud software services for DI.

Systems for Lay Users: Another promising direction is to customize DI systems discussed so far for lay users (who do not know how to code), by adding GUIs/wizards/scripts as a layer on top. A lay-user action on a GUI, for example, is translated into commands in the underlying system (in a way analogous to translating a Java statement into assembly code). A key challenge is to build this top layer in a way that is easy for lay users to use yet maximizes the range of tasks that they can perform.

Collaborative Systems: Similarly, we can try to extend the DI systems discussed so far to collaborative settings (including crowdsourcing). This raises many interesting challenges, e.g., how can users (who are often in different locations) collaboratively label a sample and converge to a match definition along the way? How can they collaboratively debug, or clean the data? How can power users, lay users, and possibly also crowd workers work together?

6 Integrating Research, Education, and Outreach

We now discuss how extending PyData allows us to pursue an *integrated* agenda of research, system development, education, and outreach, which in turn can position our community to be a key player in data science.

First, pursuing the above system building agenda raises numerous research challenges. Examples include how to develop the individual tools? How to make them interoperate? How to scale and handle data bigger than memory? How to manage DI workflows (e.g., scaling, incremental execution, what-if execution, provenance)? How to build cloud/collaborative/lay user systems, and many more. Research challenges in building current DI systems would still arise here. But we also have additional challenges that are unique to the PyData setting (e.g., how to make packages interoperate and how to scale across packages?).

Second, we can teach students in DS or DI classes DI principles, solutions, and tools drawn from PyData and PyDI, in a seamless fashion. This helps them gain deep knowledge about DI, but also trains them in tools that they are likely to use in later jobs. Further, many workshops that train domain scientists to do DS already teach PyData, so it is natural to incorporate teaching PyDI.

Third, to make PyDI successful, we need to work with real users and real data. A promising solution for academic researchers is to talk with *domain scientists* at the same university. Twenty years ago this might have been difficult, because most of the data was still at companies. But the situation has dramatically changed. At virtually any university now, often within a walking distance from the CS department, there are many domain science groups that are awash in data, with many pressing DI problems to solve. Since many such groups have been using PyData, we believe they would be willing to try PyDI.

Finally, given that DI is a major part of the DS pipeline, if we can do DI well, via system building, education/training, and outreach to domain sciences, our community should be in a good position to be a key player in data science, by "owning" the DI part (i.e., the data quality part) of this new field.

7 Ongoing Work at Wisconsin

We now describe ongoing work at Wisconsin based on the above agenda, and the main observations so far.

Build Systems for Core DI Problems: We found that this agenda could be used to effectively build a variety of DI systems. Back in 2014 we spent a year building an initial version of Magellan, a Java-based stand-alone EM system, following common practice: the system translates (GUI/command-based) user actions into a workflow of pre-defined operators, then optimizes and executes the workflow. We had serious difficulties trying to extend the system in a clean way to cope with the messiness of real-world EM tasks, where iterations/subjective decisions are the norm, and where exploratory actions (e.g., visualizing, debugging) are very common but it is not clear where to place them in the translate-optimize-execute-workflow paradigm.

Once we switched to the current agenda, these difficulties cleared up. We were able to proceed quickly, and to flexibly extend Magellan in many directions [9]. Using PyData allowed us to quickly add a rich set of capabilities to the system. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports. Besides EM, we found that the same methodology could also be used to effectively build systems for attribute value normalization and string similarity joins. As discussed earlier, we are also currently working on managing missing values and cleaning tables.

Fostering PyDI: We have been working on position and experience papers [9, 18], instructions on how to develop PyData packages, datasets, and challenge problems. We have also been partnering with Megagon Labs to encourage a community around BigGorilla, a repository of data preparation and integration tools [4]. The goal of BigGorilla is to foster an ecosystem of such tools, as a part of PyData, for research, education, and practical purposes.

Build Cloud/Collaborative/Lay User Versions of DI Systems: We have developed CloudMatcher, a cloud EM service that is well suited for lay users and crowd workers (e.g., when relying solely on crowdsourcing, it can match tables of 1.8M-2.5M tuples at the cost of only $57-65 on Mechanical Turk) [14]. This is based on our earlier Corleone/Falcon work [13, 7]. We are also developing cloud services for browsing large tables and for collaborative labeling.

Integrating Research, Education, and Outreach: Research-wise we have developed a solution for debugging the blocking step of EM [24] and matching rules [28], using deep learning to match textual and dirty data [27], scaling up EM workflows [7], and developing a novel architecture for cloud EM services [14], among others. We have designed and taught two "Introduction to Data Science" courses, at undergraduate and graduate levels, in multiple semesters. In both courses, students are asked to perform multiple end-to-end DI scenarios, using real-world data and PyData tools, including our PyDI tools. For example, the Magellan system has been used in 5 classes so far, by 400+ students. An extensive evaluation of Magellan by 44 students is reported in [17]. For outreach, Magellan has been successfully used in five domain science projects at UW-Madison (in economics, biomedicine, environmental science [18, 21, 30, 1]), and at several companies (e.g., Johnson Control, Marshfield Clinic, Recruit Holdings [4], WalmartLabs). For example, at WalmartLabs it improved the recall of a deployed EM solution by 34%, while reducing precision slightly by 0.65%. The cloud EM service CloudMatcher [14] is being deployed in Summer 2018 at American Family Insurance for their data science teams to use.

Data Science: Building on the above efforts, in the past few years we have worked to become a key player in data science at UW-Madison, focusing on *data quality* challenges. Specifically, we have been working on developing an UW infrastructure (including on-premise and cloud tools) to help UW scientists solve DI challenges, designing DS courses and programs (which incorporate DI), training UW scientists in DI, providing consulting services in DI, and working with domain sciences (as described above).

Overall, we have found that the proposed agenda is highly promising, synergistic, and practical. As a small example to illustrate the synergy: it is often the case that we conducted research to develop a how-to guide for solving a DI scenario, used it in a class project to see how well it worked for students and refined it based on feedback, then used the improved how-to guide in our work with domain scientists, who can provide further feedback, and so on.

8 Conclusions

In this paper we have discussed the limitations of current DI systems, then proposed an integrated agenda of research, system building, education, and outreach, which in turn can position our community to become a key player in data science. Finally, we have described ongoing work at Wisconsin, which shows the promise of this agenda. More details about this initial work can be found at sites.google.com/site/anhaidgroup.

References

[1] M. Bernstein et al. MetaSRA: normalized human sample-specific metadata for the sequence read archive. *Bioinformatics*, 33(18):2914–2923, 2017.

[2] P. A. Bernstein and L. M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, 2008.

[3] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[4] C. Chen et al. Biggorilla: An open-source ecosystem for data preparation and integration. In *IEEE Data Eng. Bulleting. Special Issue on Data Integration*, 2018.

[5] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[6] X. Chu et al. Distributed data deduplication. In *PVLDB*, 2016.

[7] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[8] D. Deng et al. The data civilizer system. In *CIDR*, 2017.

[9] A. Doan et al. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, 2017.

[10] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[11] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool, 2015.

[12] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[13] C. Gokhale et al. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[14] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. In *BIGDAS*, 2017.

[15] J. Heer et al. Predictive interaction for data transformation. In *CIDR*, 2015.

[16] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[17] P. Konda et al. Magellan: Toward building entity matching management systems. In *VLDB*, 2016.

[18] P. Konda et al. Performing entity matching end to end: A case study. 2016. Technical Report, `http://www.cs.wisc.edu/~anhai/papers/umetrics-tr.pdf`.

[19] S. Krishnan et al. PALM: machine learning explanations for iterative debugging. In *HILDA*, 2017.

[20] S. Kruse et al. Efficient discovery of approximate dependencies. In *PVLDB*, 2018.

[21] E. LaRose et al. Entity matching using Magellan: Mapping drug reference tables. In *AIMA Joint Summit*, 2017.

[22] G. Li. Human-in-the-loop data integration. In *PVLDB*, 2017.

[23] G. Li et al. Crowdsourced data management: A survey. In *ICDE*, 2017.

[24] H. Li et al. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, 2018.

[25] C. Lockard et al. CERES: distantly supervised relation extraction from the semi-structured web. In *CoRR*, 2018.

[26] A. Marcus et al. Crowdsourced data management: Industry and academic perspectives. In *Foundations and Trends in Databases*, 2015.

[27] S. Mudgal et al. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[28] F. Panahi et al. Towards interactive debugging of rule-based entity matching. In *EDBT*, 2017.

[29] O. Papaemmanouil et al. Interactive data exploration via machine learning models. In *IEEE Data Engineering Bulletin*, 2016.

[30] P. Pessig. Entity matching using Magellan - Matching drug reference tables. In CPCP Retreat 2017. `http://cpcp.wisc.edu/resources/cpcp-2017-retreat-entity-matching`.

[31] R. Pienta et al. Visual graph query construction and refinement. In *SIGMOD*, 2017.

[32] F. Psallidas et al. Smoke: Fine-grained lineage at interactive speed. In *PVLDB*, 2018.

[33] T. Rekatsinas et al. Holoclean: Holistic data repairs with probabilistic inference. In *PVLDB*, 2017.

[34] S. W. Sadiq et al. Data quality: The role of empiricism. In *SIGMOD Record*, 2017.

[35] R. Singh et al. Synthesizing entity matching rules by examples. In *PVLDB*, 2017.

[36] M. Stonebraker et al. Data curation at scale: The Data Tamer system. In *CIDR*, 2013.

[37] X. Wang et al. Koko: A system for scalable semantic querying of text. In *VLDB*, 2018.

[38] D. Xin et al. Accelerating human-in-the-loop machine learning: Challenges and opportunities. In *CoRR*, 2018.

[39] M. Yu et al. String similarity search and join: a survey. In *Frontiers Comput. Sci.*, 2016.