Desbordante: from benchmarking suite to high-performance science-intensive data profiler

George Chernishev
Saint Petersburg State University
Saint Petersburg, Russian Federation
Unidata LLC
Saint Petersburg, Russian Federation
chernishev@gmail.com

Michael Polyntsov
Saint Petersburg State University
Saint Petersburg, Russian Federation
polyntsov.m@gmail.com

Anton Chizhov
Saint Petersburg State University
Saint Petersburg, Russian Federation
anton.i.chizhov@gmail.com

Kirill Stupakov
Saint Petersburg State University
Saint Petersburg, Russian Federation
kirill.v.stupakov@gmail.com

Ilya Shchuckin
Saint Petersburg State University
Saint Petersburg, Russian Federation
shchuckinilya@gmail.com

Alexander Smirnov
Saint Petersburg State University
Saint Petersburg, Russian Federation
alexander.a.smirnovv@gmail.com

Maxim Strutovsky
Saint Petersburg State University
Saint Petersburg, Russian Federation
Unidata LLC
Saint Petersburg, Russian Federation
strutovsky.m.a@gmail.com

Alexey Shlyonskikh
Saint Petersburg State University
Saint Petersburg, Russian Federation
shlyonskikh.alexey@gmail.com

Mikhail Firsov
Saint Petersburg State University
Saint Petersburg, Russian Federation
mikhail.a.firsov@gmail.com

Stepan Manannikov
Saint Petersburg State University
Saint Petersburg, Russian Federation
manannikov.st@gmail.com

Nikita Bobrov
Saint Petersburg State University
Saint Petersburg, Russian Federation
nikita.v.bobrov@gmail.com

Daniil Goncharov
Saint Petersburg State University
Saint Petersburg, Russian Federation
goncharov.y.daniil@gmail.com

Ilia Barutkin
Saint Petersburg State University
Saint Petersburg, Russian Federation
ilia.d.barutkin@gmail.com

Vadim Yakshigulov
Saint Petersburg State University
Saint Petersburg, Russian Federation
vadim.iakshigulov@gmail.com

Vladislav Shalnev
Saint Petersburg State University
Saint Petersburg, Russian Federation
vladislav.a.shalnevv@gmail.com

Kirill Muraviev
Saint Petersburg State University
Saint Petersburg, Russian Federation
kirill.i.muravev@gmail.com

Anna Rakhmukova
Saint Petersburg State University
Saint Petersburg, Russian Federation
rakhmukova.anna@gmail.com

Dmitriy Shcheka
Saint Petersburg State University
Saint Petersburg, Russian Federation
dmitriy.v.shcheka@gmail.com

Anton Chernikov
Saint Petersburg State University
Saint Petersburg, Russian Federation
chernikov.a.anton@gmail.com

Yakov Kuzin
Saint Petersburg State University
Saint Petersburg, Russian Federation
yakov.s.kuzin@gmail.com

Michael Sinelnikov
Saint Petersburg State University
Saint Petersburg, Russian Federation
michael.a.sinelnikov@gmail.com

Grigorii Abrosimov
HSE University
Saint Petersburg, Russian Federation
grigorii.k.abrosimov@gmail.com

Dmitriy Popov
Independent
Moscow, Russian Federation
popov.dmitriy.ivanovich@gmail.com

Artem Demchenko
Saint Petersburg State University
Saint Petersburg, Russian Federation
artem.e.demchenko@gmail.com

Sergey Belokonny
Saint Petersburg State University
Saint Petersburg, Russian Federation
belokoniy@gmail.com

Liana-Iuliia Soloveva
Saint Petersburg State University
Saint Petersburg, Russian Federation
soloveva.iul@gmail.com

Yaroslav Kurbatov
Saint Petersburg State University
Saint Petersburg, Russian Federation
yaroslav.a.kurbatov@gmail.com

Mikhail Vyrodov
Saint Petersburg State University
Saint Petersburg, Russian Federation
mikhail.v.vyrodov@gmail.com

Arthur Saliou
Saint Petersburg State University
Saint Petersburg, Russian Federation
arthur.salio@gmail.com

Eduard Gaisin
Saint Petersburg State University
Saint Petersburg, Russian Federation
edu.gaisin@gmail.com

Kirill Smirnov
Saint Petersburg State University
Saint Petersburg, Russian Federation
kirill.k.smirnov@gmail.com

Abstract

Pioneering data profiling systems — such as Metanome and OpenClean — brought public attention to science-intensive data profiling. This type of profiling aims to extract complex patterns, such as data dependencies, data constraints, association rules, and others. However, these tools are research prototypes rather than production-ready systems.

To address this drawback, we have developed Desbordante — a science-intensive, high-performance and open-source data profiling tool implemented in C++. To the best of our knowledge, Desbordante is currently the only profiler which possesses these three characteristics. Unlike similar systems, it is built with an emphasis on industrial use, and is up to an order of magnitude faster than similar systems, while requiring up to three times less memory.

Desbordante aims to open industrial-grade primitive discovery to a broader audience, focusing on domain experts that are not IT professionals. Aside from discovery of various types of patterns, Desbordante offers pattern validation, which not only reports whether a given instance of a pattern holds or not, but also points out what prevents it from holding. Next, Desbordante lets users employ its functionality from Python scripts. Together with other Python libraries, it enables developing ad-hoc solutions for data deduplication, data cleaning, anomaly detection, and other data quality problems.

In this paper, we present Desbordante, the vision behind it, and its use-cases. To provide a more in-depth perspective, we discuss its current state, architecture, and design decisions it is built on. As a consolidation paper, it synthesizes more than six years of development work, integrating findings from numerous studies to provide a comprehensive overview.

CCS Concepts

• **Information systems** → **Data mining**; *Data management systems*; • **Applied computing** → **Physical sciences and engineering**; **Enterprise computing**; • **Computing methodologies** → *Machine learning*.

Keywords

Data Mining, Data Profiling, Pattern Extraction, Data Analysis, Knowledge Discovery, Data Exploration, Anomaly Detection, Data Wrangling

ACM Reference Format:

George Chernishev, Michael Polyntsov, Anton Chizhov, Kirill Stupakov, Ilya Shchuckin, Alexander Smirnov, Maxim Strutovsky, Alexey Shlyonskikh, Mikhail Firsov, Stepan Manannikov, Nikita Bobrov, Daniil Goncharov, Ilia Barutkin, Vadim Yakshigulov, Vladislav Shalnev, Kirill Muraviev, Anna Rakhmukova, Dmitriy Shcheka, Anton Chernikov, Yakov Kuzin, Michael Sinelnikov, Grigorii Abrosimov, Dmitriy Popov, Artem Demchenko, Sergey Belokonny, Liana-Iuliia Soloveva, Yaroslav Kurbatov, Mikhail Vyrodov, Arthur Saliou, Eduard Gaisin, and Kirill Smirnov. 2024. Desbordante: from benchmarking suite to high-performance science-intensive data profiler. In *8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD) (CODS-COMAD Dec '24), December 18–21, 2024, Jodhpur, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3703323.3703725

CODS-COMAD Dec '24, December 18–21, 2024, Jodhpur, India

ACM ISBN 979-8-4007-1124-4/24/12
https://doi.org/10.1145/3703323.3703725

1 Introduction

According to [3], data profiling is the "set of activities and processes to determine the metadata about a given dataset". Metadata can be useful for data exploration, various tasks related to data quality, database management and database reverse engineering. It also has many applications [4, 19] in data integration and query optimization domains.

We divide data profiling into naive and science-intensive. Naive profiling involves extracting basic information such as the number of rows and columns in a table, identifying the minimum and maximum values in a column, detecting atomic column data types and so on. This class is well-represented, with hundreds of available tools provided by major information system vendors. Numerous open-source tools exist as well, with YData Profiling [9] being one of the most prominent examples.

On the other hand, science-intensive profilers focus on extraction of complex metadata, which requires use of special algorithms. Examples of metadata discovery are: extraction of all kinds of functional dependencies from tables, both exact and relaxed [10], association rule mining [5], detection of semantic column types [18], discovery of data constraints [7, 46] and many more. Tools that offer such profiling are much rarer. Two notable systems that provide such functionality are Metanome [27] and OpenClean [25].

In this paper, we present Desbordante (Spanish for *boundless*) — a *science-intensive*, *high-performance* and *open-source* data profiling tool implemented in C++. To the best of our knowledge, Desbordante is currently the only profiler that combines these three distinct features.

Desbordante is inspired by Metanome, but differs from it in various key points. First of all, the focus of Desbordante is on industrial application, as extracting complex metadata requires sophisticated algorithms that are highly computationally expensive.

Second, we have a different vision of use-cases for such a tool. We envision our primary users as domain experts who possess a large amount of data that they would like to explore. At the same time, these experts are not necessarily IT professionals. Our users are interested in discovering various patterns within their data which reveal non-trivial facts. These patterns are formally described through various *primitives*. In essence, a primitive is a formal description of a rule that holds over data (or a part of it) expressed via mathematical methods. Functional dependencies are a well-known example of primitives.

The experts who could be interested in pattern discovery are:

(1) Bioinformatics researchers, chemists, geologists, and, in fact, almost any scientist working with large amounts of data, especially data obtained experimentally.
(2) Financial and business analysts, salespeople, traders.
(3) Data scientists, data analysts, machine learning specialists.

For scientists who work with large amounts of data, finding an instance of a primitive indicates the presence of some pattern. Using this instance, they may be able to formulate a hypothesis, or even draw conclusions immediately, provided there is enough data. At the very least, the found pattern can provide a direction for further study. For example, the bioinformatics team of the JetBrains lab has employed such primitives in their research [43].

In the case of financial data, the researcher can also try to obtain some kind of hypothesis. There are also more mundane and in-demand applications [13, 32, 35], such as cleaning errors in data, finding and removing near duplicates, and performing data repairs, among others. Scientists may also be interested in this functionality, albeit to a much lesser extent.

As for machine learning, found primitives can help [16, 31, 47] in feature engineering and choosing the direction for ablation studies.

Furthermore, the needs of traditional users of these primitives — database administrators and engineers — are also addressed. For these professionals, identified patterns can help with defining (or recovering) primary and foreign keys, as well as setting up (or checking) all kinds of integrity constraints [12, 17, 30].

The academic database community has created a large amount of primitives describing many different patterns that may be present in data. For example, the class of relaxed functional dependencies [10] alone has over 30 distinct formulations. Every year novel primitives continue to appear. However, these primitives are largely unknown to people outside of the community. In the worst case, they simply remain a theoretical result, and at best they exist in the form of a little-known prototype that is not ready for industrial application. There are several tools [25, 27] that contain collections of algorithms that discover and validate primitives, but they are not production-ready either. These tools are performance-bound, and fail to cater to specific needs of our users as they lack the required functionality.

Therefore, the idea of Desbordante is to make these primitives accessible to the general public and give everyone the opportunity to study their data.

Now, turning to Desbordante itself:

• Desbordante offers means for both discovery and validation of primitives, with the latter featuring descriptive explanations for primitive instances that do not hold. It is available in form of command line and web interfaces, alongside Python bindings. This variety, combined with extensive customization, ensures it meets the needs of all categories of users.

• Industrial focus is expressed through efficiency and scalability. In contrast to other science-intensive profilers, the kernel of Desbordante is implemented in C++ and heavily optimized, leading to a speed up of up to 10x [24, 38] over the state-of-the-art science-intensive profilers, along with up to 3x RAM savings.

Speeding up primitive discovery is not just for sport. As we mention in Section 3.2, primitive discovery is a computationally challenging problem. Therefore, optimizing implementation is the only way to bring primitives closer to practical use.

• Python bindings allow users to employ [11] Desbordante alongside standard data scientist tools such as pandas [44]. This allows users to develop their custom programs that can include discovery and validation of primitives. As a result, it is possible to develop ad-hoc solutions to various data quality problems, e.g. data deduplication, data cleaning, anomaly detection, and so on. Furthermore, Desbordante provides "deep" Python integration, allowing the offload of various resource-intensive operations to the C++ core: not just pattern discovery, but also tasks such as providing explanations and outlier discovery.

Desbordante is an open-source project[1] developed with a modern technological stack. It has been in development for more than six years, during which we recognized the true value of primitives and transitioned from the idea of benchmarking primitive discovery algorithms [42] to building a full-fledged profiler. Currently, Desbordante has a growing userbase — since its first release in December 2023, Desbordante pip package has been installed more than 16K times, while core repository has gained about 400 stars on GitHub during the same period.

2 Related Work

It is hard to classify existing tools in such a way that they would exactly match with Desbordante by properties, functionality, or vision. Therefore, we review most well-known tools which implement various data profiling techniques and can mine primitives, or at least highly rely on them.

The closest relative of Desbordante in the data profilers family is Metanome [27]. In a nutshell, Metanome is a framework which provides developers with an infrastructure and a corresponding set of interfaces for implementing and benchmarking primitive-related algorithms. Metanome's architecture makes the process of developing and testing research ideas as fast and convenient as it can be, thus enabling developers to concentrate on the algorithms instead of boilerplate code for database connectors, text processing utilities, etc. Metanome was developed in Hasso Plattner Institute and almost any algorithm developed by the same group can be plugged into Metanome as a JAR file. However, Metanome can not be considered as a true industrial alternative to Desbordante due to the reasons we present in an extensive evaluation and comparison of both platforms [42], which include larger memory footprint and inferior dependency discovery speed just to name a few.

Furthermore, its vision is different: it considers discovered primitive instances as the final result. However, we believe that these instances should be treated as building blocks that can be used for further data analysis and manipulations. In our view, a production-ready tool makes primitives available to practitioners. From this standpoint, Metanome lacks the ability to include found instances in Python programs (Python interface), primitive validation and explanations, and combine primitives into pipelines.

The goal of the OpenClean [25] system is to become a part of a modern data science stack by taking on the niche of data cleaning and profiling. Being an open-source Python library, OpenClean provides its users with an environment in which they can seamlessly integrate data profiling with other frameworks and libraries for data processing and machine learning. Consider the concept of functional dependencies (FDs). In OpenClean it can be used in two ways: checking data for FD violations and FD mining. The former is implemented in Python as a combination of mapper and group filters (much like the SELECT…GROUP BY …HAVING idiom for FD checking in SQL). For records which violate the FD, a repair process can be started using either the OpenClean or user-defined

[1] https://github.com/Desbordante/desbordante-core

repair strategies. The primitive mining functionality is provided by a standalone package which initiates a subprocess for running Metanome JAR files. Basically, any algorithm that was once implemented for Metanome can be run within OpenClean. It also means that, in terms of algorithm performance, OpenClean inherits all problems of Metanome.

Unlike OpenClean, most data cleaning tools have no built-in functionality for primitive mining, and expect the user to provide primitives as input data to the cleaning process [8, 13, 33, 34]. For example, a data repairing framework HoloClean [33] makes good use of denial constraints (DC), which subsumes functional dependencies (FD), conditional FDs (CFDs) and other FD-related concepts. The input of HoloClean is an inconsistent dataset, a set of DCs and any external knowledge which can be used to repair the dataset. HoloClean combines every piece of available information and proposes solutions that can bring the data to a consistent state. Since the main goal of HoloClean is to pave the road to a careful restoration of the dataset to a consistent form, the tool does not implement any internal mechanisms for mining of primitives or for efficient in-place inference of metadata. The same vision is shared by Horizon [34], which computes an FD pattern graph based on the FD set provided, and constructs a solution via pattern graph static analysis.

However, not every data profiling tool considers primitives to serve as great foundation for a cleaning process or error detection task. Those tools instead rely on machine learning, probabilistic methods [45], or a curated knowledge base like Katara [14]. The authors of Katara even refuse to consider FDs as trusted metadata, since this type of primitive can not guarantee that data would be fixed in a non-ambiguous way.

Some data engineering tools follow a different philosophy: instead of fixing data, they make sure it is consistent in the first place. Great Expectations [2] allows its users to define complex integrity constraints, which are used as an assertion mechanism while the data flows through ETL processes. Such tiny unit tests for data validation can be embedded into a workflow and immediately raise a flag if anything unexpected happens, e.g., newly arrived data violates a primitive that was described within the Great Expectations framework. A similar idea is used in the Auto-Validate [39] system.

The aforementioned tools are implementations of research findings which are carefully surveyed in dozens of papers, with some of these tools being open-source and free to use. To make the overview complete, we would like to list some industrial solutions.

Most commercial solutions provide support for the concept of FD as part of data profiling: SAP Information Steward, Oracle Warehouse Builder, Informatica Data Quality, Microsoft SQL Server Data Profiling Task, and Talend Open Studio can return functional dependencies which almost hold on data, or verify whether a user-specified dependency holds. For each AFD, these tools also maintain the fraction of records that violate a dependency. It seems that this way of processing FD/AFD is almost a must-have feature for any data profiling tool and it comes in handy during anomaly detection and exploring "broken" records. However, usually FD/AFD are the only types of primitives that are implemented in a pay-to-use tool, since nowadays their focus has shifted to the machine learning side of the data profiling spectrum.

3 Discovery of science-intensive primitives

3.1 Current State and Motivation

A **primitive** is a mathematical description of a rule (pattern) that may or may not **hold** over data. Functional dependencies [28] are a good example: dependency $A \longrightarrow B$ (A and B being columns) holds if for each pair of rows in a table it is true that from the equality of values in A follows the equality of values in B.

There are several hundred types of primitives [10, 41], and each of them has well-established properties and a sound theory behind it. Furthermore, the development of new types of primitives is an ongoing process.

However, as stated previously, they largely stay within the database and associated communities, and provide no benefit to a broader public. The reasons for this are the following:

• Largely, implementations of primitive-related algorithms are poorly accessible or not available at all:

- The majority of them were developed in the pre-GitHub era, when it was not customary for authors to provide source code, or the source code was published on the research group's website, which is usually long gone by now. Either way, currently there is no source code available.
- Those that are currently accessible are scattered around the Web, on personal websites, or in obscure repositories. Of course, the presenting paper usually includes a link, but prospective users have to know about the primitive and the paper first, which is usually not the case.

• If accessible, these tools are hard to set up and run. For example, it took the newcomers of our team from 6–12 work hours to set up and run Metanome, despite their technical backgrounds and familiarity with IT specifics. Thus, for someone without IT expertise wishing to experiment with a primitive, it could be a challenging task.

• Each available implementation of a primitive (or even a single discovery algorithm) would require its own software ecosystem to set up and maintain. It is another obstacle to overcome for a prospective non-technical user who would like to try some primitives.

• Finally, available implementations often serve as initial proofs of concept or prototypes which were developed for academic purposes and not actively maintained afterwards. Therefore, they are usually not very efficient, since they were developed in a language which favours rapid prototyping, like Java or Python. These languages lack efficiency and low-level tunability of C++. Additionally, scalability concerns arise, given that real world datasets are likely to be larger than those benchmarked by researchers. Thus, it is necessary to move the limits of applicability further (hence the name *Desbordante*).

There are platforms which try to address these issues, such as Metanome or OpenClean. However, they fail to address all of them simultaneously. Thus, there is a need for an industrial-grade platform which will open primitive discovery to a broader public, and Desbordante tries to achieve this goal.

3.2 Specifics of science-intensive primitive discovery

First of all, primitive discovery is a computationally hard problem as it belongs to the forefront of science. Discovery algorithms run into

time or memory limits even when processing small datasets. This is illustrated in Table 1 from [29], which shows that the majority of datasets suitable for functional dependency mining on server-class hardware are smaller than one megabyte. The situation is similar in case of other primitives. In order to make the discovery of primitives truly usable in practice, we need to address these limitations.

Furthermore, there are aspects that are specific to the vision and goals of our system. They reflect use-cases and needs of our users.
1) Our users are more interested in approximate primitives. Real world data is likely to have all kinds of errors, missing values, and other types of artifacts. Consequently, exact versions of primitives are not always applicable, since they are rarely found in real data. Instead, we must provide inexact versions that allow a certain degree of error.
2) Our users need not only the discovery of primitives, but also their validation. Unlike discovery, validation accepts a specific instance of a primitive (e.g. a specific functional dependency) as input and returns whether it holds or not. This leads to the need for providing explanations, i.e. what prevents a given primitive from holding (e.g. conflicting values, rows, etc.). These explanations will greatly help with outlier discovery and data analysis in general.
3) Our users need to be provided with various tuning knobs for the discovery process. For example, concerning the discovery of functional dependencies, it is well-known that dependencies with a larger left-hand side are less valuable. Their discovery usually does not indicate the presence of a real dependency, but instead points to the fact that the data fragment which was used for mining is too small to contain a counterexample. Primitive discovery process is always costly and it is worthwhile to skip unnecessary computations. Another important example is setting the error threshold for approximate primitives. There are no "universal" thresholds, as correct values depend on the particular dataset and user goals. Therefore we have to give users a convenient way to experiment with those values.
4) Our users have different preferences with regards to the interface. Some of them prefer an old-school command-line interface, while others ask for a rich web UI. Furthermore, providing a Python interface is essential to open up primitives to data scientists and domain experts. This will allow experimentation with primitives and the construction of custom data analysis pipelines. The ultimate goal here is to enable the development of ad-hoc solutions to various data quality problems, such as data deduplication, data cleaning, anomaly detection, and more.

Desbordante is designed to take into account these specifics.

4 Desbordante

4.1 Overview

Primitives are defined on different types of data, be it a single table, multiple tables, or graphs. For the primitives defined on the data types named, Desbordante currently supports the following tasks:
• **Discovery**: find all holding instances of the specified primitive over a dataset.
• **Validation**: given a primitive instance, determine whether it holds over the specified dataset. If it does not, provide additional information on why it does not hold.

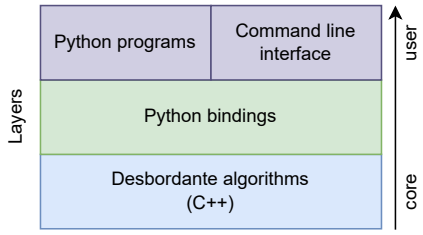


Figure 1: Desbordante's interfaces and their use

For the primitives defined on a single table, the platform also contains algorithms that are optimized for the case where data can change over time. We call these algorithms "dynamic", because data is updated *dynamically*. Classic algorithms are thus referred to as "static". Where a static algorithm would have to reprocess the whole table again, its dynamic counterpart can instead use the new data to perform updates on the previously acquired result, making it much faster in certain situations.

Desbordante also supports naive profiling by allowing the extraction of basic statistics from a table, such as the median or sum of all values in a column. However, that is not the focus of the platform, so we will omit the discussion of this functionality.

Desbordante's core is a C++ library (see Fig 1) containing all auxiliary data structures needed for primitive discovery and validation algorithms, the algorithms themselves, and all of the supporting infrastructure. There are multiple ways in which users can interact with the core library.

The first one is the `desbordante` Python library. It works seamlessly with pandas' `DataFrames` and provides an interface that lends itself well to both scripting and use in Python's interactive mode.

The general way of using a static algorithm is as follows:

(1) Create the appropriate algorithm object.
(2) Initialize it with data by calling the `load_data` method.
(3) Configure and execute it with the `execute` method.
(4) Obtain the results using the appropriate method.

An example of usage is provided in Fig. 2.

```
import desbordante

# Select the algorithm for mining AFDs
algo = desbordante.afd.algorithms.Default()

# Configure the algorithm with CSV table data
algo.load_data(table=('path/to/table.csv', ',', True))

# Set the error threshold (discard AFDs with a
# larger error) and execute the algorithm
algo.execute(error=0.1)

# Retrieve and print the discovered AFDs
print('AFDs:')
for fd in algo.get_fds():
    print(fd)
```

Figure 2: Example code for mining approximate FDs

The workflow is slightly different in case of dynamic algorithms. The initial results on the unchanged table can already be obtained

after the `load_data` call, and the `execute` method is used to process table updates. A usage example is presented in Fig. 3. More detailed dynamic algorithms usage (with inserting/updating table rows) can be found in the Desbordante's example folder mentioned below.

```
import desbordante
import pandas as pd

# Select the algorithm for dynamic FD verification
algo = desbordante.dynamic_fd_verification.\
    algorithms.Default()

# Read a CSV file into pandas DataFrame
data = pd.read_csv('path/to/table.csv', header=[0])

# Configure the algorithm with the DataFrame,
# specify column indices to verify the FD [0, 2] -> 1
# and start the FD verification algorithm
algo.load_data(table=data, lhs_indices=[0, 2],
               rhs_indices=[1])

# Print whether the FD holds on the original table
print(f"FD holds: {algo.fd_holds()}")

# Define set of row indices to delete
delete_set = {0, 4, 5}
# Update the table by deleting the specified indices
# and recalculate FD state (whether it holds)
algo.execute(delete=delete_set)

# Print whether the FD holds after the update
print(f"FD holds: {algo.fd_holds()}")
```

Figure 3: Example code for dynamic FD verification

The Python library is the primary interface that users are meant to utilize when working with Desbordante.

The second way in which users can interact with the core library is the command line interface, which is built on top of the Python library using Click [1]. It allows users to choose which algorithm to run and configures it using the provided parameters.

There is also the third interface — the Web-interface. Its deployed version is available at the link[2]. Unfortunately, due to space constraints, we have to omit its discussion in this paper.

We recognize that the exact definitions of primitives can be difficult to understand, and the ways of applying them may not be immediately apparent, thus raising the barrier of entry. Desbordante addresses this issue by providing a set of examples in the form of Python programs[3] that demonstrate the primitives and their usage.

Overall, examples can be divided into three groups:

• **Basic** examples showcase a single primitive by discussing its definition and providing a concise introduction using simple examples. Furthermore, they briefly describe how to use it — that is, how to discover or validate it in Python code. We provide at least one example for each supported primitive.

• **Advanced** examples illustrate various nuances, for example those concerning primitive differences. It is assumed that the user is familiar with these patterns and studied the basic examples to understand examples from the advanced section.

• **Expert** examples demonstrate instances of complex programs which can be built using primitive discovery or primitive validation functionality offered by Desbordante. These programs aim to provide tangible benefits for end-users by solving real-life problems.

4.2 Primitive Discovery

The first class of tasks Desbordante is aimed at is primitive discovery. It goes as follows: for a given dataset, find all holding instances of a specified primitive.

Desbordante strives to offer as much tunability of primitive discovery as it can. Every algorithm has its own set of supported parameters. The basic ones define which type of primitive to discover and the data on which to operate, as well as some filters on the desired primitive instances.

Let us proceed to discussing some important parameters.

• Threshold — the degree of allowed violations, for algorithms supporting approximate primitives.

• Search Depth — limits discovery depth to save time when not all primitive instances are needed. Discovery process is time-consuming and the user doesn't always need all of the instances.

• Number of Threads — specifies how many threads are used for discovery or validation in multi-threaded algorithms.

Desbordante tries to provide multiple algorithms for each supported primitive. For some primitives, the best performing algorithm is more or less known, but for some it is not. For example, a particular discovery algorithm may perform badly on a "wide" or a "long" table. It may also run out of available memory on a particular dataset, since different algorithms are built upon different principles, and the specifics of the dataset largely affect the algorithm's memory consumption. Therefore, we have decided to provide several algorithms for each primitive.

The console interface for the primitive discovery task is rather mundane. On the contrary, the Python interface is quite interesting, as it makes the results of primitive discovery available in Python programs, allowing users to manipulate found primitive instances in a variety of ways. For example, it is possible to perform set-based operations, such as intersection or union, as well as object comparisons.

Note that found primitive instance might not necessarily hold in general, as it could be discovered because the dataset failed to contain a counterexample. Therefore, our users will need primitive validation (e.g. on a larger dataset), which we will now describe.

4.3 Primitive Validation and Explanations

Primitive validation is the second class of tasks Desbordante is aimed at. It is defined as follows: for a given primitive instance and a dataset, find out whether a specific primitive instance holds or not. However, if it does not hold, users need explanations and information on what prevents it from holding. It is essential to provide that information, since it is an important piece of knowledge about the data that is being explored by the user. It can indicate errors in data and point out "problematic" records, i.e. it can help with outlier discovery. Therefore, there is a need for a way to provide it inside the tool.

[2]https://desbordante.unidata-platform.ru

[3]https://github.com/Desbordante/desbordante-core/tree/main/examples

Explanations are primitive-specific, as they depend on the definition of each individual primitive. However, each of them provides some context — records, calculated thresholds, distances and so on.

An example of explanations for exact functional dependency [28] validation is presented in Fig. 4. The upper part of the figure contains data (a table describing game characters and their parameters) which is being studied. A user validates $Creature \rightarrow Strength$ dependency, which does not hold. Below, an explanation is provided. It shows clusters of records that share the same left hand side (lhs), but differ in the right hand side (rhs). There are two such examples — one in the "Elf" and another in the "Ogre" entities. "Dwarf" table fragment is fine, this particular dependency holds. Suppose that a user knows that this dependency should hold. In the "Elf" case there are three records and two conflicting values, therefore it is reasonable to think that this "1" is a typo, as it is outvoted. The "Ogre" cluster is more tricky, some outside domain knowledge is required to resolve this contradiction. Finally, the context for this primitive includes a number of distinct rhs values and the proportion of the most frequent rhs value for each cluster.

In the figure, we decided to display found clusters and related information. However, in Python programs all this data is fully accessible. Having requested it from Desbordante, a user can manipulate it in a variety of ways while employing numerous third-party Python libraries. For example, they can select and sort clusters and/or records within clusters using various parameters such as distance, index, number of outliers and so on. Access to this information significantly contributes to the development of automatic tools for developing ad-hoc solutions for data deduplication, data cleaning, anomaly detection, and many other data quality problems.

```
   Creature  Strength  HaveMagic
0      Ogre         9      False
1      Ogre         6      False
2       Elf         6       True
3       Elf         6       True
4       Elf         1       True
5     Dwarf         9      False
6     Dwarf         9      False

FD Creature -> Strength does not hold.
Number of clusters violating FD: 2
 #1 cluster:
2: Elf -> 6
3: Elf -> 6
4: Elf -> 1
Most frequent rhs value proportion: 0.67
Num distinct rhs values: 2

 #2 cluster:
0: Ogre -> 9
1: Ogre -> 6
Most frequent rhs value proportion: 0.5
Num distinct rhs values: 2
```

Figure 4: Explanation for FD validation task

At the same time, requesting data from Desbordante is done in an efficient way, which we refer to as Deep Python integration. This will be discussed next.

4.4 Deep Python integration

We refer to the ability to use the platform alongside standard data scientist tools like pandas [44] as Python integration. The workflow with Python bindings has been described above: first, the user selects an algorithm by creating the algorithm object in Python. The user then configures it, executes it, and retrieves results through appropriate methods specific to the type of the algorithm.

Desbordante's tabular data processing algorithms accept pandas DataFrames as input, allowing the user to conveniently preprocess data before mining primitive instances. The results of an algorithm can be converted to basic Python types like lists and tuples, so it is easy to perform additional filtering and other further processing techniques with the powerful facilities Python provides.

As for the "deep" part, this is to contrast Desborante's "deep" integration to the "shallow" integration offered by OpenClean. In this context, "deep" refers to the ability to offload various resource-intensive operations — not just pattern discovery, but also tasks such as generating explanations and finding outliers — to the C++ core. This conceptual difference in approaches in shown in Fig. 5. For example, in OpenClean validation of a functional dependency is implemented in Python as a combination of mapper and group filters (much like the `SELECT...GROUP BY...HAVING` idiom for functional dependency checking in SQL), while in Desbordante it can be done natively with a single call to the C++ library. Ultimately, Desbordante's deep Python integration minimizes the need to run significantly less efficient Python code and spares end-users from having to implement such functionalities themselves.

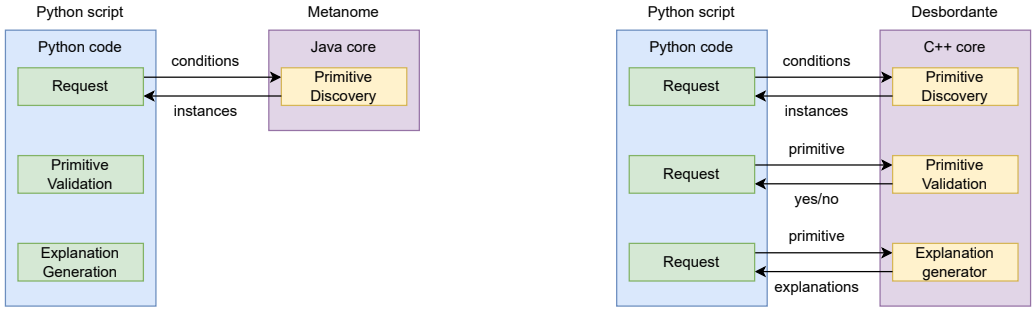
4.5 User scenarios

The Python ecosystem contains a large variety of libraries suited for many data analysis and processing tasks, making it a principal tool for data scientists. Consequently, enabling users to process discovered primitive instances within the same environment is essential. Given this context, the Desbordante Python interface is designed to offer an easy method to extract these instances and their associated data directly from the profiler core. This facilitates seamless processing of discovered data. Overall, our approach enables users to experiment with primitives and construct ad-hoc data analysis pipelines.

Recently, the MDedup [23] system highlighted the usefulness of primitive discovery for solving data quality problems. Following this, we believe it is necessary to facilitate rapid development of applications that use primitives. Examples of such functionality, implemented using Desbordante, are available at the provided link[4], with the Streamlit versions accessible through link[5]. We describe the ideas behind some of these examples in our demo paper [11].

Overall, Desbordante scenarios will enable reimplementation and ensure the repeatability of plethora of existing approaches to performing data repairs, as proposed in many existing papers [13, 21, 35], as well as the future ones. As a result, such approaches can be applied in practice and with less effort.

[4]https://github.com/Desbordante/desbordante-core/tree/main/examples/expert
[5]https://desbordante.streamlit.app/



(a) Shallow Python integration (OpenClean) **(b) Deep Python integration (Desbordante)**

Figure 5: Architectural approaches to organizing Python integration

4.6 Performance

Unlike all existing open-source solutions, the discovery part of Desbordante is fully implemented in modern C++. While Python and Java are generally simpler and facilitate faster development, they come with a number of no less prominent drawbacks:
(1) Given equal effort put into code, the resulting performance of Java/Python applications is worse than that of C++ on average.
(2) Java apps usually leave a higher memory footprint than C++.
(3) Finally, these languages restrict opportunities for low-level optimizations, such as vectorization via SIMD instructions. This is a critical drawback for solving a high-performance computing task.

At the same time, modern C++ provides plenty of room for low-level optimizations. For example, for inclusion and order dependency discovery, the speedup compared to existing Java implementations reached up to ten times [24, 38]. In these cases, we have used a simple vectorization approach, custom hash tables, and profiled code. For algorithms that were simply reimplemented without optimization, the speedup is usually up to three times [42].

Note that speeding up primitive discovery is not just for sport. As mentioned in Section 3.2, primitive discovery is a computationally challenging problem. Therefore, optimizing implementation is the only way to bring primitives closer to practical use.

Another key advantage of C++ implementation is the decrease in memory consumption. Our experiments demonstrated up to three times RAM savings [24, 37, 38]. This is crucial, since many primitive discovery algorithms are memory-bound [29], therefore reducing memory footprint enables processing of larger datasets.

It is important to note that we have not fully exhausted the tuning potential of the C++ implementation. Currently, Desbordante uses default STL and Boost data structures, and we have not tuned their parameters. Desbordante does not rely on custom memory management libraries (allocators) either, but instead uses the C++ default. It is a well-known fact [26] that using a special allocator is a simple yet efficient way to improve performance of C++ programs. Therefore, it is possible to improve performance and reduce memory consumption even more.

4.7 Industrial Focus

Industrial focus is primarily expressed through performance and scalability. Furthermore, Desbordante comes with a modern codebase and rigorous development process: unit-testing, continuous integration, extensive use of static checkers and linters, mandatory code reviews among others. Finally, Desbordante features a rich set of available interfaces (web, console, Python), numerous examples and provides overall easiness to deploy via Pip packages. The latter allows to install Desbordante via a single command.

5 Supported primitives

Due to the reasons stated in Section 3, Desbordante provides a slightly different set of primitives than Metanome. The full list of primitives supported by Desbordante can be found on the website[6]. Overall, there are about 20 supported primitives and more than 30 implemented algorithms. Furthermore, several features are not included in the latest release yet but are already available in development branches, such as the discovery of matching dependencies [36], the discovery of soft functional dependencies [20], and the validation of numeric dependencies [15].

For now, we lack a number of primitives that Metanome has, such as conditional inclusion dependencies, denial constraints and others. However, we provide other types that are absent in it (e.g. metric FDs [22] or Differential Dependencies [40]) that are more relevant for our use-cases. To the best of our knowledge, for some of primitive types, such as graph functional dependencies [17], metric FDs, and algebraic constraints, our implementation is the only one publicly available. Finally, in the future we plan to greatly expand the number of primitives and catch up — some of the missing primitives are already in the works.

6 Evaluation

In order to evaluate our system, we considered two kinds of metrics:
• **Popularity.** Desbordante offers one-of-a-kind functionality to a broader audience, which can serve as a basis for data exploration, hypothesis generation, development of a new generation of data-quality tools, and much more. Despite being in development and lacking many significant primitives, our tool is already popular.

[6] https://github.com/Desbordante/desbordante-core

Table 1: Desbordante primitives

Primitive	Avg. speed up	Max. speed up	Avg. Memory Improvement	Techniques Applied
Functional Dependencies	17.67x	47.41x	not measured	Parallelization
Approximate FDs (g_1)	2.12x	3.43x	1.88x	None
Order Dependencies	5.25x	10x	2.6x	Algorithmic, Data Structures
Matching Dependencies	69.73x	170x	20x	Algorithmic, Data Structures, Parallelization
Unique Column Combs	17x	47x	$\geq$ 10x	Parallelization
Inclusion Dependencies	3.75x	8x	not measured	Buffering, SIMD, Data Structures, Parallelization

Since the initial release, our Python package was installed more than 16K times[7] and the core repository gained about 400 stars.

• **Performance.** Desbordante is superior to its alternatives in terms of processing speed and required memory. The profiler includes dozens of algorithm implementations, described in individual papers [6, 24, 38, 42] and reports[8], each containing their own evaluation. Due to space constraints, it is virtually impossible to present a comprehensive evaluation. Instead, we will describe the general picture using Table 1. Here, we list only some of the primitives as we are short on space and for many primitives Desbordante is the only one available implementation. In the table we provide the obtained speedup, memory savings, and techniques, employed to achieve this speedup. Among these techniques are: 1) Algorithmic — a modification of algorithmic nature, 2) Buffering — bulk data processing in order to achieve cache-conscious algorithm behavior, 3) Data Structures — selection and usage of more efficient data structures, 4) Parallelization — usage of multi-threading, 5) SIMD — usage of vectorization. We compared algorithms for discovery of primitives with their counterparts from Metanome and other prototypes. Note that we included Pyro (approximate FD discovery), to illustrate the fact that even straightforward reimplementation (without any additional effort) would still yield speedup.

Acknowledgments

This project is the result of a joint effort of a community spanning more than 50 people. Unfortunately, we are tight on space to list them all, therefore we present the most important contributors in the author list. In addition, we would like to mention the following people for their significant contribution to the project (in alphabetical order): Pavel Anosov, Bulat Biktagirov, Maksim Emelyanov, Maxim Fofanov, Igor Korobitsyn, Vladislav Makeev, Pyotr Senichenkov, Egor Shalashnov, Alexander Slesarev, Nikita Talalay, Ivan Volgushev. The complete list of project participants can be obtained in the GitHub repositories of the Desbordante organization. Finally, we would also like to thank Anna Smirnova for her help with preparation of this paper.

7 Conclusion

In this paper we have presented Desbordante — a one-of-a-kind open-source data profiler with the focus on discovery of complex patterns in data. Desbordante aims to open industrial-grade pattern discovery to a broader public, focusing on domain experts who are not IT professionals. Unlike similar systems, it is built with emphasis on industrial application: it is efficient and flexible, provides both pattern discovery and validation, while offering explanations for the latter. It comes with three interfaces: command-line interface, web interface, and Python bindings. The latter is the most important one as it offers means for efficient usage of pattern discovery and validation in Python programs. This will allow users to experiment with patterns and develop ad-hoc solutions to various data quality problems, e.g. data deduplication, data cleaning, anomaly detection, and so on. Overall, we hope that Desbordante will make primitives first-class entities in data analysis and that primitive-based data analysis will gain widespread adoption.

[7] https://www.pepy.tech/projects/desbordante

[8] https://github.com/Desbordante/desbordante-core/tree/main/docs/papers

References

[1] [n. d.]. Click home page. https://click.palletsprojects.com/.

[2] James Campbell Abe Gong. [n. d.]. Great Expectations. https://github.com/great-expectations/great_expectations.

[3] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling Relational Data: A Survey. *The VLDB Journal* 24, 4 (aug 2015), 557–581. https://doi.org/10.1007/s00778-015-0389-y

[4] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling.* Morgan & Claypool Publishers.

[5] Charu C. Aggarwal and Jiawei Han. 2014. *Frequent Pattern Mining.* Springer Publishing Company, Incorporated.

[6] Ilia Barutkin, Maxim Fofanov, Sergey Belokonny, Vladislav Makeev, and George Chernishev. 2024. Extending Desbordante with Probabilistic Functional Dependency Discovery Support. In *2024 35th Conference of Open Innovations Association (FRUCT).* 158–169. https://doi.org/10.23919/FRUCT61870.2024.10516409

[7] Leopoldo Bertossi, Loreto Bravo, Enrico Franconi, and Andrei Lopatenko. 2008. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Information Systems* 33, 4 (2008), 407–434. https://doi.org/10.1016/j.is.2008.01.005 Selected Papers from the Tenth International Symposium on Database Programming Languages (DBPL 2005.

[8] George Beskales, Ihab F. Ilyas, and Lukasz Golab. 2010. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 197–207. https://doi.org/10.14778/1920841.1920870

[9] Simon Brugman. 2019. pandas-profiling: Exploratory Data Analysis for Python. https://github.com/pandas-profiling/pandas-profiling.

[10] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2016. Relaxed Functional Dependencies—A Survey of Approaches. *IEEE Transactions on Knowledge and Data Engineering* 28, 1 (2016), 147–165. https://doi.org/10.1109/TKDE.2015.2472010

[11] George A. Chernishev, Michael Polyntsov, Anton Chizhov, Kirill Stupakov, Ilya Shchuckin, Alexander Smirnov, Maxim Strutovsky, Alexey Shlyonskikh, Mikhail Firsov, Stepan Manannikov, Nikita Bobrov, Daniil Goncharov, Ilia Barutkin, Vladislav Shalnev, Kirill Muraviev, Anna Rakhmukova, Dmitriy Shcheka, Anton Chernikov, Mikhail Vyrodov, Yaroslav Kurbatov, Maxim Fofanov, Sergei Belokonnyi, Pavel Anosov, Arthur Saliou, Eduard Gaisin, and Kirill Smirnov. 2023. Solving Data Quality Problems with Desbordante: a Demo. *CoRR* abs/2307.14935 (2023). https://doi.org/10.48550/ARXIV.2307.14935 arXiv:2307.14935

[12] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proc. VLDB Endow.* 6, 13 (aug 2013), 1498–1509. https://doi.org/10.14778/2536258.2536262

[13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[14] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: Reliable Data Cleaning with Knowledge Bases and Crowdsourcing. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1952–1955. https://doi.org/10.14778/2824032.2824109

[15] Paolo Ciaccia, Matteo Golfarelli, and Stefano Rizzi. 2013. Efficient derivation of numerical dependencies. *Information Systems* 38, 3 (2013), 410–429. https://doi.org/10.1016/j.is.2012.07.007

[16] Vargha Dadvar, Lukasz Golab, and Divesh Srivastava. 2022. Exploring data using patterns: A survey. *Inf. Syst.* 108, C (Sept. 2022), 11 pages. https://doi.org/10.1016/j.is.2022.101985

[17] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional Dependencies for Graphs. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1843–1857. https://doi.org/10.1145/2882903.2915232

[18] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César Hidalgo. 2019. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1500–1508. https://doi.org/10.1145/3292500.3330993

[19] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. Association for Computing Machinery, New York, NY, USA.

[20] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 647–658. https://doi.org/10.1145/1007568.1007641

[21] Sijia Jiang, Zijing Tan, Jiawei Wang, Zhikang Wang, and Shuai Ma. 2023. Guided conditional functional dependency discovery. *Information Systems* 114 (2023), 102158. https://doi.org/10.1016/j.is.2022.102158

[22] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric Functional Dependencies. In *2009 IEEE 25th International Conference on Data Engineering*. 1275–1278. https://doi.org/10.1109/ICDE.2009.219

[23] loannis Koumarelas et al. 2020. MDedup: Duplicate Detection with Matching Dependencies. *Proc. VLDB Endow.* 13, 5 (jan 2020), 712–725.

[24] Yakov Kuzin, Dmitriy Shcheka, Michael Polyntsov, Kirill Stupakov, Mikhail Firsov, and George Chernishev. 2024. Order in Desbordante: Techniques for Efficient Implementation of Order Dependency Discovery Algorithms. In *2024 35th Conference of Open Innovations Association (FRUCT)*. 413–424. https://doi.org/10.23919/FRUCT61870.2024.10516381

[25] Heiko Müller, Sonia Castelo, Munaf Qazi, and Juliana Freire. 2021. From Papers to Practice: The Openclean Open-Source Data Cleaning Library. *Proc. VLDB Endow.* 14, 12 (oct 2021), 2763–2766. https://doi.org/10.14778/3476311.3476339

[26] "No Bugs" Hare. 2018. Testing Memory Allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc While Trying to Simulate Real-World Loads. http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/

[27] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1860–1863. https://doi.org/10.14778/2824032.2824086

[28] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (June 2015), 1082–1093. https://doi.org/10.14778/2794367.2794377

[29] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 821–833. https://doi.org/10.1145/2882903.2915203

[30] Thorsten Papenbrock and Felix Naumann. 2017. A Hybrid Approach for Efficient Unique Column Combination Discovery. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (LNI, Vol. P-265)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.). GI, 195–204. https://dl.gi.de/handle/20.500.12116/628

[31] Frédéric Pennerath, Panagiotis Mandros, and Jilles Vreeken. 2020. Discovering Approximate Functional Dependencies using Smoothed Mutual Information. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1254–1264. https://doi.org/10.1145/3394486.3403178

[32] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern functional dependencies for data cleaning. *Proc. VLDB Endow.* 13, 5 (jan 2020), 684–697. https://doi.org/10.14778/3377369.3377377

[33] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[34] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-Driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (oct 2021), 2546–2554. https://doi.org/10.14778/3476249.3476301

[35] Rashed Salem and Asmaa Abdo. 2016. Fixing rules for data cleaning based on conditional functional dependency. *Future Computing and Informatics Journal* 1, 1 (2016), 10–26. https://doi.org/10.1016/j.fcij.2017.03.002

[36] Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient Discovery of Matching Dependencies. *ACM Trans. Database Syst.* 45, 3, Article 13 (Aug. 2020), 33 pages. https://doi.org/10.1145/3392778

[37] Alexey Shlyonskikh, Michael Sinelnikov, Daniil Nikolaev, Yurii Litvinov, and George Chernishev. 2024. Lightning Fast Matching Dependency Discovery with Desbordante. In *2024 36th Conference of Open Innovations Association (FRUCT)*. 729–740. https://doi.org/10.23919/FRUCT64283.2024.10749955

[38] Alexander Smirnov, Anton Chizhov, Ilya Shchuckin, Nikita Bobrov, and George Chernishev. 2023. Fast Discovery of Inclusion Dependencies with Desbordante. In *2023 33rd Conference of Open Innovations Association (FRUCT)*. 264–275. https://doi.org/10.23919/FRUCT58615.2023.10143047

[39] Jie Song and Yeye He. 2021. Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1678–1691. https://doi.org/10.1145/3448016.3457250

[40] Shaoxu Song and Lei Chen. 2011. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.* 36, 3, Article 16 (Aug. 2011), 41 pages. https://doi.org/10.1145/2000824.2000826

[41] Shaoxu Song, Fei Gao, Ruihong Huang, and Chaokun Wang. 2022. Data Dependencies Extended for Variety and Veracity: A Family Tree. *IEEE Transactions on Knowledge and Data Engineering* 34, 10 (2022), 4717–4736. https://doi.org/10.1109/TKDE.2020.3046443

[42] Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, and George Chernishev. 2021. Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms. In *2021 29th Conference of Open Innovations Association (FRUCT)*. 344–354. https://doi.org/10.23919/FRUCT52173.2021.9435469

[43] Petr Tsurinov, Oleg Shpynov, Nina Lukashina, Daria Likholetova, and Maxim Artyomov. 2021. FARM: Hierarchical Association Rule Mining and Visualization Method. In *Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics* (Gainesville, Florida) *(BCB '21)*. Association for Computing Machinery, New York, NY, USA, Article 70, 1 pages. https://doi.org/10.1145/3459930.3469499

[44] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*. 56 – 61.

[45] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. 2013. Don't Be SCAREd: Use SCalable Automatic REpairing with Maximal Likelihood and Bounded Changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 553–564. https://doi.org/10.1145/2463676.2463706

[46] C.T. Yu and W. Sun. 1989. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering* 1, 3 (1989), 362–375. https://doi.org/10.1109/69.87981

[47] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 861–876. https://doi.org/10.1145/3318464.3389749