Large-Scale Data Pollution with Apache Spark

Kai Hildebrandt, Fabian Panse, Niklas Wilcke, and Norbert Ritter

Abstract—Because of the increasing volume of autonomously collected data objects, duplicate detection is an important challenge in today's data management. To evaluate the efficiency of duplicate detection algorithms with respect to big data, large test data sets are required. Existing test data generation tools, however, are either not able to produce large test data sets or are domain-dependent which limits their usefulness to a few cases. In this paper, we describe a new framework that can be used to pollute a clean, homogeneous and large data set from an arbitrary domain with duplicates, errors and inhomogeneities. To prove its concept, we implemented a prototype which is built upon the cluster computing framework Apache Spark and evaluate its performance in several experiments.

1 Introduction

To integrate data from multiple heterogeneous sources or to clean a single database, duplicates need to be detected [1], [2], [3]. Duplicate detection is the task of identifying data objects that refer to the same real-world entity [4], [5], [6]. Because of heterogeneous representations (e.g. different data formats, languages or units of measurements) as well as missing, erroneous and outdated values, such an identification is a tough challenge [7]. The result of a duplicate detection process is a partition of the considered set of data objects (the so-called *duplicate clustering*) where each partition class (i.e. duplicate cluster) represents another real-world entity [8].

To evaluate the quality of a newly developed duplicate detection algorithm or to find the parameter setting which is the best fit for a given application scenario, we require an application-specific data set that is labeled with its corresponding solution (the so-called *gold standard* [8]). Such a pair of data set and gold standard is called *test data set* through the rest of this paper. Figure 1 presents an unclean database relation (in this case data objects correspond to database tuples) and its gold standard which in this example is modeled by an additional attribute containing the cluster-id of each tuple.

According to Christen [6], test data sets can be acquired in five different ways:

- by reusing the results of a former duplicate detection process (ideally from the same domain),
- by labeling the objects of a real-life data set (semi-)manually with the help of domain experts,
- by using one of the few test data sets, such as CDDB or Cora, which were available in the web[1,2],
- by synthesizing an unclean data set from scratch, or
- by polluting a clean (real-life) data set with duplicates, errors and inhomogeneities.

In times of big data, duplicate detection algorithms have to deal with large sets of data objects and hence have to be efficient in terms of runtime and storage requirements [9], [10]. To evaluate such efficiencies, we need test data sets of large sizes. Manual labeling is an extremely time consuming task and hence can only be applied to small data sets. Public test data sets typically result from manual labeling and therefore are small, too (e.g. the CDDB and Cora data sets each contain less than ten thousand objects). Moreover, reusing the results of former duplicate detection processes is usually not an option because we do not have any test data to evaluate the quality of

ID	*forename*	*surname*	*size*	*sex*	*cluster-id*
1	Bill	Hlal	176	m	1
2	Meg	Lee	171	0	2
3	Jule	Smith	174	f	3
4	Megan	Lee	171	f	2
5	Paul	Ryan	186	m	4
6	Hall	Bill	178	m	1
7	Will	Hall	5.84	m	1

Fig. 1. Test data set with seven objects and four duplicate clusters.

these results. Finally, existing tools for data synthesization and data pollution, such as DBGen [7], the Febrl data set generator [11], GeCo [12], TDGen [13] or ProbGee [14] are either not able to generate sets with more than one million objects, are limited to a specific schema (and hence domain), and/or the resultant data sets are far away from being realistic.

For these reasons, we identified a set of desiderata that should be met by a test data generation tool with respect to big data and developed a data pollution framework which meets these desiderata. This framework consists of three phases. In the first phase, a homogenous clean input data set is read and analyzed by using data profiling techniques. Important characteristics of this data set (e.g. the inferred data type, the percentage of NULL values and the degree of uniqueness per attribute) are stored in an attribute-based data profile. In the second phase, this profile is used to automatically derive a set of application-specific representation and error schemas. Finally, based on these schemas, the clean input set is corrupted with data errors including duplicates. In each of these three phases, the framework is designed in a way that its workload can be distributed on the different nodes of a computer cluster. To prove its concept, we implemented a prototypical distributed data pollution tool called *DaPo* that is based on this framework and built upon the cluster computing framework Apache Spark[3] [15]. We evaluated this tool with regard to the individual desiderata where we focused on the scalability aspect. The experimental results show that this prototype scales with an increasing number of cluster nodes and hence enables the generation of large test data sets in an acceptable time on the condition that a sufficiently large computer cluster is given.

The contributions of this paper are:

- an analysis of desiderata for test data generation tools,
- a survey on existing tools (synthesization and pollution) for generating test data sets for duplicate detection,
- a novel approach for iterative error injection,
- a novel framework for data pollution with an automatic preconfiguration of schema-dependent parameters,
- a prototypical data pollution tool based on Apache Spark,
- an experimental evaluation of this prototype.

• *All authors are with the University of Hamburg, Germany.*
E-mail: {6hildebr,panse,1wilcke,ritter}@informatik.uni-hamburg.de

1. http://hpi.de/naumann/projects/repeatability/datasets
2. http://www.cs.utexas.edu/users/ml/riddle/data.html

3. http://spark.apache.org/

The remainder of the paper is structured as follows: First we identify a set of desiderata which are (more or less) vital for an appropriate test data generation tool in Section 2. Then, we survey existing tools for test data generation in the field of duplicate detection in Section 3. Thereafter, we introduce our novel data pollution framework (including the novel error injection approach) in Section 4. In Section 5, we present DaPo and discuss several experiments performed with this prototype in Section 6. Finally, we conclude this paper and give an outlook on open challenges in Section 7.

2 Desiderata

In this section, we define a set of desiderata that should be met by a useful test data generation tool with respect to big data.

2.1 Efficiency & Scalability

Since we aim to produce test data sets that contain millions (or even billions) of objects, the primary desiderata is that the designed tool is able to generate large data sets in an acceptable runtime. To accomplish this goal, it has to work efficiently. Moreover, it is useful if the tool scales vertically and/or horizontally so that an increasing data size can be treated with the addition of extra hardware resources.

2.2 Schema Independence

It does not make sense to use test data about personal information in order to find an appropriate configuration setting for a duplicate detection process that should work in the domain of molecules. As a consequence, it is important that the test data generator is independent from a specific schema, and thus can be used to produce test data of an arbitrary domain.

2.3 Realistic Data Values & Patterns

The insights that are gained from evaluating a test data set are only useful if the values and value patterns of this set are realistic for the considered application scenario. Therefore, each attribute should only contain values belonging to its domain (e.g. no name of a person should contain a digit). Important value patterns include (but are not restricted to):

- integrity constraints such as uniqueness, domain restrictions or functional dependencies,
- frequencies of single values, attribute value combinations and NULL values.

Note that integrity constraints address the error-free state of the data set and therefore can be violated in the generated test data set, but such violations should always result from errors (e.g. a name value ‘t1m’ which results from an OCR-error) or duplicates (e.g. two books with the same ISBN). In general, whether or not an integrity constraint should be violated by the pollution process depends on the assumptions made on the simulated data sources. For example, if a source is assumed to be a database whose schema models all these constraints, none of them should be violated by the polluted data objects. In contrast, if a source is assumed to be a simple text file or a database whose schema is poorly modeled, constraint violations are part of realistic error patterns (see next desiderata).

2.4 Realistic & Variable Error Patterns

Not only the values and value patterns of the generated data set need to be realistic, but this condition also holds for the injected errors and the patterns that emerge between these errors.

Whereas some errors are typical for many domains (e.g. typos), there are several errors that only emerge in specific domains (e.g. measurement errors). For this reason, the generation of realistic errors requires a large range of error classes [2], [16], [17], [18] which should be extendable to new classes easily.

In reality, data is oftentimes copied from one source to another [19]. Thus, data errors are often spread among sources belonging to the same network. In addition, errors are often produced during data collection so that software (e.g. transformation or calculation errors) and hardware bugs (e.g. measurement errors) as well as specific software (e.g. application-specific drop-down lists), and hardware conditions (e.g. specific keyboard layouts) of the collection process can affect multiple data objects in the same way. Moreover, data is often outdated and because outdated values were correct once, it can be assumed that different objects provide the same outdated values even if they originate from different sources. In summary, in realistic test data sets it is not unusual that different (duplicate) data objects contain the same or similar errors (e.g. the same typos, measurement errors or outdated values), especially if these objects were created by the same method (e.g. person, tool or methodology) and are managed by the same party.

Finally, data representation is typically source-specific (i.e. all objects of the same source are represented in the same way) so that heterogeneous representation forms (e.g. different data formats or units of measurement) produced by the data generation process should not be completely random, but follow some meaningful pattern.

2.5 Simple but Adaptable Configuration

The generated test data should fit a specific application scenario. Therefore, it is important that the generation process can be configured in a flexible way. Nevertheless, users are often unwilling to spend too much time in configuring a system. In conclusion, we aim for a data generation tool that requires little configuration effort, but provides many configuration options.

3 Related Work

In this section, we survey existing tools for relational[4] test data generation that are used in the context of duplicate detection. The first two are data synthesization tools and the last two are data pollution tools. The third tool can be used in both ways.

3.1 DBGen

The UIS database generator (DBGen) [7] was developed in 1997 (a second version was published 1999). DBGen starts with generating synthetic personal data tuples by using look-up tables and rules, and then pollutes these tuples with duplicates and errors. DBGen has a rudimentary GUI, but can also be used via command line.

3.1.1 Schema & Flexibility

The schema of the generated data set is hard-coded and contains twelve attributes (including a cluster-id for the gold standard) which describe various personal characteristics like names, dates of birth or addresses. Although the source code (written in C) is open, an adaptation to another schema would result in considerable work and would require competent knowledge on the functionality of the generator.

3.1.2 Duplicate Generation

The number of injected duplicates is specified as a percentage of the number of synthetically generated tuples (each a duplicate cluster). Moreover, a cluster size distribution (uniform, Poisson or Pareto) can be selected by the user.

4. Tools for generating test data sets of non-relational data formats such as the Dirty XML Data Generator (http://hpi.de/naumann/projects/completed-projects/dirtyxml.html) are not included.

3.1.3 Error Injection

The error injection mechanism used in DBGen is strongly adjusted to the considered person schema. It is hard-coded which error classes can be applied to which attributes. The provided error classes include typos, transpositions of fore- and surnames, substitutions of forenames with initials, deletions of middle names, substitutions of surnames based on look-up tables, substitutions of street name endings (e.g. 'street', 'avenue' or 'st'), or substitutions of whole addresses.

3.1.4 Limitations, Efficiency & Scalability

DBGen is really fast and requires little memory. Its runtime scales linearly with the number of generated tuples so that even large data sets can be created in short time. However, large generated data sets are far apart from being realistic because the provided look-up tables are small (ca. 2,500 names and ca. 2,100 ZIP-City-State triples) and the same look-up table is used for first-, last- and street names which results in a low data diversity. Moreover, DBGen does not consider dependencies between the individual attributes (except the ones between ZIP, City and State) and does not use realistic distributions of attribute values. Although the user can increase diversity by incorporating more and larger look-up tables, a general observation remains: The more tuples are generated, the less realistic becomes the generated data set.

DBGen is neither multi-threaded nor distributed and hence always runs on a single core. At least, even with larger look-up tables (500 times greater than the provided ones) only the constant effort of loading these tables into memory increases so that DBGen is also very fast in such cases. By using the larger look-up tables, the generation of a data set with one million tuples took less than five seconds.

3.2 Febrl Data Set Generator

The Febrl data set generator (short FebrlDG) [11], [18] was developed by Peter Christen and Agus Pudjijono between 2005 (Version 1) and 2008 (Version 2) and is part of the duplicate detection framework Febrl (Freely extensible biomedical record linkage) of the Australian National University. This tool follows a very similar approach as DBGen and first generates synthetic data tuples modeling personal information based on look-up tables as well as rules, and then corrupts these tuples with errors, heterogeneous formattings and duplicates. The generated tuples, however, are much more realistic than the ones produced by DBGen because more and larger look-up tables are used. Furthermore, the generation process takes relative frequencies of possible attribute values as well as dependencies between specific attributes into account. This tool consists of a source-open python script and can be used via command line.

3.2.1 Schema & Flexibility

The schema of the generated database relation contains 18 attributes (including information on the gold standard) and is hard-coded in the aforementioned python script. Since python is a script language and python code does not need to be compiled, a modification of the schema is simpler than in DBGen, but detailed knowledge about the source code is still necessary. Thus, depending on the amount of intended schema modifications, the adaptation effort can still be large.

3.2.2 Duplicate Generation

The user specifies the number of original tuples and duplicates. As in DBGen, the user can choose between a uniform, a Poisson and a Pareto distribution to determine the individual cluster sizes. In addition, a maximal cluster size can be defined.

3.2.3 Error Injection

FebrlDG supports a variety of error classes such as typographical, phonetical, OCR and misspelling errors, insertions and deletions of blanks, deletions and overwritings of attribute values, transpositions of string tokens within the same attribute value and transpositions of values between different tuples within the same attribute [18]. The python script contains a probability for each combination of attribute and error class which can be modified manually (see discussion above).

3.2.4 Limitations, Efficiency & Scalability

Principally, FebrlDG can be used to generate (almost) large data sets in an acceptable runtime. Due to the sophisticated and large look-up tables, these sets are much more realistic than these generated with DBGen. Unfortunately, this tool is neither multi-threaded nor distributed. Thus, it runs on a single processor core and the maximal size of the generated data set is limited by the amount of available main memory. In our experiments, runtime increased linear with the number of generated tuples and generating a test data set with one million tuples took 253 seconds. On a machine with 8 GB RAM, the maximal test data set size was three million tuples.

3.3 GeCo

GeCo (Data Generator and Corruptor) [12], [20] was developed 2012 at the Australian National University as a successor of FebrlDG. As its predecessor, GeCo generates synthetic data tuples and corrupts them with errors and duplicates. However, there are several differences between both tools. Instead of a single python script, GeCo consists of multiple python modules which can be integrated into an own project. Moreover, GeCo has a web interface[5] which can be used to generate small data sets with up to 9,999 tuples [20]. In contrast to its predecessor, GeCo cannot be controlled via command line. In GeCo, the generation of a test data set is divided into two step. In the first step, a duplicate-free set of synthetic tuples is generated. In the second step, errors and duplicates are injected into this set. Actually, this corresponds to the approach used in DBGen and FebrlDG, but because the process is divided into two steps and the second step can be executed stand-alone, the user is enabled to import an own (maybe real-life) data set as input to the corruption step. Finally, GeCo is able to handle different Unicode character sets and hence is able to process non-latin characters.

3.3.1 Schema & Flexibility

Whereas DBGen and FebrlDG are based on a hard-coded schema designed for storing personal data, GeCo enables the user to define an own schema (nevertheless, GeCo is especially useful for personal data). If only a small data set is required, the easy to use web interface fits very well. For generating large data sets, however, the user needs to code a python program which uses the provided GeCo modules to define a generation and/or corruption process.

3.3.2 Duplicate Generation

The duplicate generation process corresponds to this of FebrlDG (see Section 3.2).

5. ANU Online Personal Data Generator and Corruptor: https://dmm.anu.edu.au/geco/

3.3.3 Error Injection

In GeCo, the process of error injection can be configured at a very detailed level. The user is able to set the number of errors per tuple, the number of errors per attribute value and the probability that a specific error is injected into the values of a specific attribute. In addition, a list of possible error classes along with respective probabilities can be defined for each attribute. The configuration process is time-consuming and has to be done via the web interface or by using python dictionaries and lists. The provided error classes essentially correspond to those that are provided by FebrlDG.

3.3.4 Limitations, Efficiency & Scalability

GeCo is able to generate test data sets based on a given input set, has many features, can be configured in various ways and is well documented. As its predecessor, GeCo is neither multi-threaded nor distributed and the maximal size of the generated test data set is limited by the amount of available main memory. The runtime of GeCo increases almost linear with the number of generated tuples, but is generally very slow. Thus, GeCo is only suitable for a generation of large data sets if long runtimes from several hours up to multiple days are acceptable for the user. In our experiments, GeCo needed 13.5 hours to corrupt a test data set with one million tuples (i.e. processing only included the corruption step).

3.4 TDGen

TDGen was developed by the German Record Linkage Center at the University of Duisburg-Essen in 2012 [13]. It is conceptually based on FebrlDG, but is restricted to the injection of errors into a given data set and hence cannot generate duplicates by its own. TDGen is implemented as an extension of the analytics java platform KNIME[6] and provides a predefined error injection workflow. By default, TDGen is used via the KNIME GUI, but (as every KNIME workflow) it can also be executed via command line.

3.4.1 Schema & Flexibility

Principally, TDGen is schema-independent because it can process any data set which is provided in form of a CSV-file. In practice, however, it turned out that the use of custom data sets can require considerably adaptations of the TDGen workflow, due to issues regarding data types and NULL values. Otherwise, TDGen can produce errors which lead to an abort of the whole process.

3.4.2 Duplicate Generation

An explicit duplicate generation is not provided by TDGen. Therefore, generating a test data set containing duplicates and a gold standard, requires a preparation of the input data set by another tool which inserts exact duplicates whose values are later polluted by TDGen.

3.4.3 Error Injection

The main reason for developing TDGen was to get a more flexible control of the degree and way data is polluted than it is provided by FebrlDG. In TDGen, the user can adapt the predefined TDGen workflow to her own needs or can create an all-new one. Within the workflow, several parameters can be adjusted. The error injection process can be controlled separately for rows, columns and fields by defining maximal numbers of errors and error probabilities. In overall, the error injection process can be controlled very precisely, but despite of a GUI its configuration is not very user-friendly.

TDGen provides a large number of error classes which for the most part are adopted from FebrlDG, but also includes some special classes for string values like 'keep the longest token' and some classes which are tailormade for specific data types such as dates or ZIP codes.

3.4.4 Limitations, Efficiency & Scalability

TDGen partly uses multiple processor cores and principally has a satisfactory runtime. However, the size of the processed data set is limited by the amount of available heap space. Since the error injection process is extremely memory-intensive, this upper limit is exceeded relatively fast.

In our experiments, TDGen was not able to process more than ten thousand tuples if we used the standard configuration. By optimizing the TDGen workflow, we were able to increase this number to one hundred thousand. In this case, TDGen required 4.5 minutes. Principally, TDGen scales only vertically. However, it seems natural to split a large data set into several small data sets and to pollute each of these sets on a separate cluster node. One approach to do this, would possibly the (commercial) KNIME plugin 'KNIME Cluster Execution' which can be executed on a cluster by using the Sun Grid Engine.

3.5 ProbGee

ProbGee [14] is a tool for deriving (probabilistic) test data sets from an existing data set which was developed at the University of Hamburg in 2012. This tool was originally designed to generate probabilistic test data. However, it can also be used to generate regular (i.e. certain) test data sets. ProbGee was implemented on the basis of the java-based Rich Client Platform of Eclipse and thus provides an Eclipse-similar GUI.

3.5.1 Schema & Flexibility

ProbGee provides some preconfigured scripts for parsing the Java Movie Database[7] which contains movie data from IMDb. In principle, however, every relational database which is provided by a CSV-file or which can be accessed via JDBC can be used as generation input. In addition, ProbGee provides a number of parameters that focus on the generation of probabilistic data and hence are not considered in this discussion.

3.5.2 Duplicate Generation

In ProbGee, the user is able to specify how many duplicate clusters of which size should be generated. To prevent that each value has to be inserted into the input mask separately, the cluster size distribution can be automatically filled by specifying some parameters and manually modified afterwards. These parameters are the number of tuples of the to be generated data set, the minimal and maximal cluster size as well as the used distribution (uniform, normal, exponential, etc.).

3.5.3 Error Injection

The error injection process of ProbGee is particularly sophisticated and highly configurable. To control the degree of pollution, the user can specify a similarity measure and an individual weight for each attribute. In addition, the user defines a so-called 'duplicate similarity' which corresponds to a target value of the weighted average similarity between the corrupted tuples of the same duplicate cluster. Furthermore, the user can specify a probability for each combination of error class and

6. https://www.knime.org/

7. http://www.jmdb.de/

TABLE 1
Characteristics of Existing Data Generation Tools

Tool	year of release	interfaces	config-urability	conf. effort (minimal)	schema/domain independence	degree of pollution	horizontal scalability	runtime (100k)	runtime (1mio)	runtime (10mio)
DBGen	1997	GUI/CLI	low	low	✗	✗	✗	5 s	7 s	25 s
FebrlDG	2008	CLI	low	low	✗	✗	✗	2 min	4.2 min	✗
GeCo	2012	Lib/Web-UI	high	high	✓	✗	✗	80 min	13.4 h	✗
TDGen	2012	GUI/(CLI)	high	medium	✓	✗	✗	4.5 min	✗	✗
ProbGee	2012	GUI	high	medium	✓	✓	✗	5.9 h	✗	✗
DaPo	2015	CLI	high	low	✓	✓	✓	31 s[1] 17 s[1*] 24 s[2] 15 s[2*]	2.1 min[1] 56 s[1*] 46 s[2] 27 s[2*]	26 min[1] 14 min[1*] 7.3 min[2] 3.4 min[2*]

[1] computed by using a single desktop computer (local mode).
[2] computed by using a computer cluster with one master and eight worker nodes (distributed mode).
[*] computed by using predefined representation/error schemas (no automatic analysis/preconfiguration phases).

attribute. For this purpose, however, the user has to manipulate the XML-files of the automatically created generation scripts.

The duplicate generation process is executed based on the previously set configuration and consists of two steps. In the first step, for each original tuple, a tree of tuple-variations (a so-called *error provenance tree*) is constructed by injecting errors randomly. In the second step, a subset of the tree's variations is selected by considering their duplicate similarity and the user-defined target value. Finally, the selected tuple variations form a duplicate cluster of the generated test data set.

In ProbGee, a variety of error classes is implemented. Many of these error classes are especially suitable for the provided JMDB data, but can be applied to any string data in general. Moreover, the architecture of ProbGee allows the user to implement and add new error classes to this open-source software. The set of provided error classes includes typos based on confusion matrices, semantic errors (e.g. synonyms) based on confusion sets, corruptions of year values, transpositions of values within the same tuple and insertions of NULL values.

3.5.4 Limitations, Efficiency & Scalability

ProbGee can derive (probabilistic) test data sets from any given data set, has many features and can be configured in various ways. Nevertheless, ProbGee is neither multi-threaded nor distributed. In fact, its runtime increases (almost) linear with the number of generated tuples, but is very poor in general. The maximal size of the generated test data set is limited by the amount of available heap space.

In our experiments, ProbGee took around six hours to generate a test data set with hundred thousand tuples. A data set with one million tuples could not be generated because the available 8 GB main memory was too small. As a consequence, for large data sets, ProbGee is principally not suitable.

3.6 Comparison & Conclusion

Some of the main characteristics of the discussed tools are summarized in Table 1 (note the characteristics of DaPo are separatelty discussed in Section 6.5). The usage of DBGen and FebrlDG is per se limited to a generation of personal data. To evaluate the runtime behavior of GeCo and TDGen, we used the same MusicBrainz data sets as for DaPo (see Section 6.1). In the case of ProbGee, we used the default movie database which consists of four regular attributes. Each test run was performed on the master node of our Spark cluster using 7GB main memory (see Section 6.2). Moreover, we configured all tools in a way that they produce similar degrees of pollution.

As one can see, among the presented tools only DBGen, FebrlDG and GeCo were able to generate test data sets with more than one million objects. DBGen, however, is extremely inflexible because it is limited to a hard-coded schema and the error injection process can be configured only to a small extent. Thus, the resultant data sets are somewhat unrealistic and cannot be adjusted to the considered application scenario. At least, the efficiency of DBGen is so outstanding that 10 million objects can be generated in a few seconds. FebrlDG suffers from similar flexibility problems than DBGen, but had much longer runtimes and could not produce a data set with 10 million objects. In contrast, GeCo has many features, is schema-independent and can be configured in various ways. However, compared to DBGen, the efficiency of GeCo is extremely weak because it already took 13.4 hours to process one million objects.

In summary, none of these tools meets all the desiderata we have listed in Section 2. Interestingly, only one of them uses more than one processor core, none of them is distributed, and almost all of them exceed their memory limits very fast.

Because of the outlined shortcomings, such as schema dependency or inefficiency (i.e. long runtimes), adapting one of these tools to a parallel and distributed execution was not an option and we decided to design a new one instead.

4 Data Pollution Framework

In this section, we present our data pollution framework by describing its instruments and workflow.

4.1 Instruments

The goal of our project was to design a test data set generator which satisfies all the desiderata we have listed in Section 2. To accomplish this, we used a number of instruments.

4.1.1 Parallelization & Distribution

A parallel and distributed execution increases efficiency. Moreover, it enables vertical and horizontal scalability because parallelization on single machines can be used to exploit all the cores of a multi-core processor (vertical scalability) and a distribution of workload across multiple machines enables exploiting the collective resources of a computer cluster (horizontal scalability). Due to the latter, hardware resources like CPU power, main and secondary memory can theoretically be up- (scale-out) or downgraded (scale-in) at pleasure.

4.1.2 Data Profiling

Data profiling [21] describes the automated analyzing of data sets. In our framework, we use attribute-based profiling for identifying several (statistical) characteristics of individual attributes and attribute combinations. Together with the schema information provided by the data source, these characteristics are stored in a data profile.

```
ErrorSchema(root=RowErrorSequence((
  ToLower(p=0.14),
  TransposeFieldsInRow(p=0.1),
  FieldErrorSchema(fields=Array(
    null,
    OneByScore(p=0.14,(Typo,Abbreviate,OCR,Phonetic)),
    OneByScore(p=0.14,(Typo,Abbreviate,OCR,Phonetic))
  ))
))
ObjectCondition(ID<=1000)
)
```

Fig. 2. Example of an error schema.

```
RepresentationSchema(root=RowTransFSequence(
  ToLower(All),
  ToDollar(Currency),
  ToFormat(Date,'YYYY-MM-DD'),
  Language(English-US))
ObjectCondition(ID mod 5 = 0)
)
```

Fig. 3. Example of a representation schema.

4.1.3 *Enriched Data Schema*

Depending on the type of the considered data source, a data schema can contain different amounts of information. For example, whereas CSV-files only provide the number and maybe the names of attributes, the schema of a relational database also contains information on data types (e.g. VARCHAR, INT or user-defined domains) and constraints (e.g. UNIQUE, NOT NULL or CHECK clauses). In this framework, an enriched data schema is used which does not only include the schema information of the input data source, but also includes other kinds of information being relevant for the later executed processing steps. This approach enables schema independence and sets a basis for generating realistic error patterns as well as realizing a system which can be configured in a flexible way.

4.1.4 *Automated Preconfiguration*

To unburden the user from spending too much configuration effort even though a large number of parameters is given, a preconfiguration of the actual data pollution process is automatically derived from the data profile by using reasoning techniques such as rules or machine learned classifiers.

4.1.5 *Error Model*

To generate realistic test data, the injected errors should fit to the given input data. For that reason, we use an error model containing the following concepts:

- **Error Classes:** Each error class models a specific kind of error and can be categorized by its scope. *Row-errors* concern single data objects, *column-errors* concern single attributes, and *field-errors* concern single attribute values. Since most field-errors are domain-specific, this class can be further divided into several subclasses. For instance, we need other types of errors for numerical values than for words or word sequences. Error classes for different domains have been extensively investigated in the database community [2], [16], [17], [18] and can be reused in our framework. Examples of field-errors are typos, OCR-errors (e.g. '1' $\leftrightarrow$ 'l'), token errors (e.g. 'list of' $\leftrightarrow$ 'list of of'), phonetic errors (e.g. 'Clyne' $\leftrightarrow$ 'Klein'), numerical errors (e.g. $120 \leftrightarrow 119$), or formatting errors (e.g. 'Jill A. Doe' $\leftrightarrow$ 'Doe, Jill Ann'). Examples of row-errors are:
  – transposing/substituting/shifting/merging the values from different attributes,
  – applying the same field-error to all values of one object (e.g. the same OCR-error or the same formatting error).

  Examples of column-errors are:
  – transposing/substituting/shifting/merging the values from different objects,
  – applying the same field-error to all values of one attribute (e.g. the same calculation error).

  As we will discuss in Section 5.2.2, the distinction of error classes by scope is helpful for a parallel and distributed execution of the pollution process on possibly partitioned data. The distinction of field-errors by domain enables a flexible handling of different data types and hence supports the generation of realistic errors.
- **Error Schemas:** An error schema models a complex procedure of error injection and is built up by combining and nesting different error classes with the help of meta-errors (e.g. sequences or choices). By assigning probabilities and weights to the individual error-classes (including meta-errors), parts of the procedure can be executed randomly. As a result, error schemas allow a flexible configuration of the pollution process. An example of an error schema is presented in Figure 2 where a sequence of row-errors is applied to all objects having an ID lower than or equal to 1000.
- **Representation Schemas:** To enable an injection of heterogeneity that is caused by integrating objects from data sources designed, filled and managed autonomously, we introduce the concept of representation schemas. Each representation schema defines a source-specific way to represent data objects. Among others, such representations can differ in their:
  – sets of attributes (e.g. the two attributes 'forename' and 'surname' vs. a single attribute 'name'),
  – languages (e.g. english (US), english (GB) or french),
  – data formats (e.g. '24.01.16' vs. '2016-01-24'),
  – data encodings (e.g. $\{0, 1\}$ vs. $\{f, m\}$ to model gender information),
  – levels of abstraction (e.g. 'New York City' vs. 'Manhattan' to model residence information),
  – units of measurements (e.g. inch vs. cm or \$ vs. €).

  Note, whereas some of these aspects are source-specific (e.g. language), some of them only concern particular attributes of this source (e.g. units of measurements). In contrast to error schemas, a representation schema does not contain probabilities and weights, but is a deterministic sequence of transformation processes. An example of a representation schema is presented in Figure 3 where the values of every object whose ID modulo 5 is 0 are transformed into lower case, its date values are transformed into the 'YYYY-MM-DD' format, its currency values are transformed into dollars and its language is changed to American English.
- **Object Groups:** Another concept that we introduce to enable a configuration of realistic errors is called *object groups*. The idea of such groups is to specify objects having an error schema in common. By doing so, we generate structural, syntactic and semantic heterogeneity between the objects of different groups in order to simulate different data models (e.g. CSV-file or relational table with predefined data types and CHECK constraints), input conditions (e.g. form or API), acquisition contexts (e.g. weather conditions or sensor quality) or acquisition times. An object group is defined by adding a boolean condition to an error schema (e.g. all objects with ID $\leq$ 1000 or CITY='Hamburg'). Because the individual conditions do

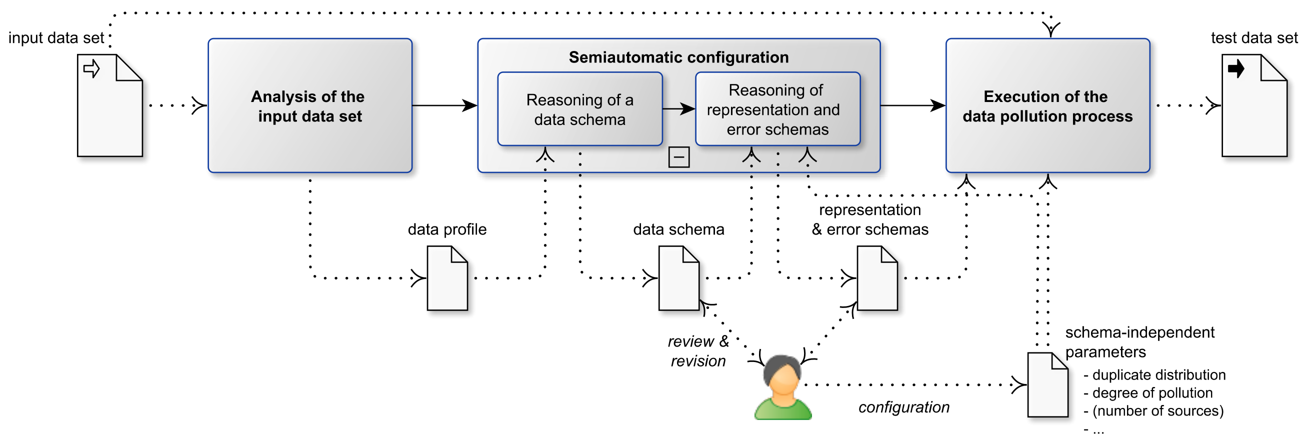

input data set
Analysis of the input data set
Semiautomatic configuration
Reasoning of a data schema
Reasoning of representation and error schemas
Execution of the data pollution process
test data set
data profile
data schema
representation & error schemas
review & revision
configuration
schema-independent parameters
- duplicate distribution
- degree of pollution
- (number of sources)
- ...

Fig. 4. Workflow of the presented data pollution framework.

not have to be exclusive, it is possible that an object belongs to more than one group. This approach can be used to reduce the number of required error schemas if different object groups have error patterns in common.

- **Source Groups:** Source groups are specific object groups that partition the set of data objects into multiple disjoint data sources. They are defined by adding boolean conditions to the representation schemas. Since every data object has a unique form of representation, the conditions of the representation schemas need to be disjoint (i.e. every object belongs to at most one source group). However, the disjunction of these conditions does not need to be a tautology, because all objects that are not covered by any of the defined representation schemas belong to an indirectly modeled source group whose representation form corresponds to the one of the input set.
- **Error Inheritance:** As described in Section 2, in real-life data, it is not uncommon that duplicate objects contain the same (or similar) errors even if they originate from different data sources (e.g. because data objects were copied across sources or values which were correct once are outdated now). In this framework, such error correlations can be injected into the polluted data objects by using the concept of error inheritance. The underlying idea of this concept is to inherit errors from a data object to one or more of its duplicate objects. The integration of this idea into the framework is described in Section 4.2.3.

4.2 Workflow

The input to our data pollution framework is a homogenous clean (and hence duplicate-free) data set. Its output is the generated test data set. The workflow of this framework is presented in Figure 4 and consists of three main phases:

- In the first phase, the homogenous and clean input set is analyzed by means of attribute-based data profiling. The analysis results are then stored in a data profile along with the schema information provided by the input source.
- To be able to execute the actual data pollution process, several parameters need to be configured in advance. This is accomplished in two steps. First, an enriched data schema is derived from the data profile and then several representation and error schemas are derived from this data schema. Each of these automatically generated schemas has to be regarded as a preconfiguration which can be reviewed and (if needed) manually revised by the user afterwards.
- In the last phase, the actual data pollution process is executed based on the configurations computed in the previous phase. This execution phase primarily consists of the injections of duplicates, inhomogeneities and errors into the input data set.

We consider each of these three phases in more detail in the rest of this section.

4.2.1 *Analysis of the Input Data Set*

In this phase, the input data set is imported and analyzed by running several data profiling algorithms. First of all, the schema information provided by the input source is imported and all useful aspects are extracted. Thereafter, for each attribute a number of (statistical) information is computed and stored in a data profile. Among others, this profile contains the following information per attribute:

- **Distribution of Basis Data Types:** Often the input file does not provide any information on data types or the provided types are very unspecific (e.g. ZIP codes are not typed as five-digit numbers, but integers or strings). For that reason, the profiling algorithm counts how often the individual basis data types fit to a value of the considered attribute. The domain of basis data types depends on the implementation, but should include at least the four types string, double, integer and boolean. In addition, the number of NULL values is stored.
- **Original Basis Data Type:** The data type provided by the input file (if available).
- **Diversity of Attribute Values:** The profiling algorithm counts the number of distinct values and computes their share in the total number of attribute values (i.e. the distinct values ratio). Moreover, it counts the number of occurrences for each of these distinct values.
- **Type-Specific Statistics:** Depending on the basis data type, additional information on the attribute's values can be useful. For example, in the case of numerical attributes, it make sense to store statistical characteristics like the minimum, maximum, average or variance of all values. In the case of string attributes, it is helpful to compute such statistical characteristics about
  - the number of tokens (e.g. words) per value,
  - the token lengths, or
  - the number of occurrences of individual letters or tokens within all values.

To enable an identification of integrity constraints such as functional dependencies, the profiling algorithm also evaluates statistics concerning more than one attribute (and stores them in the data profile).

4.2.2 Semiautomatic Configuration

The information gained from the analysis of the input set can be used to configure the schema-dependent parameters of the pollution process in a meaningful way. For this purpose, we automatically derive preconfigurations of these parameters based on the given data profile.

This phase consists of two steps. Initially, based on the data profile an enriched data schema is derived. Among others, this schema contains the following aspects:

- The number and names of all attributes.
- A basis data type per attribute which is either the original one (if available) or one of the types provided by the basis type distribution.
- An abstract data type per attribute. Abstract data types are more specific concepts like *word*, *word sequence* or *text* in order to distinguish between different string types, but can also correspond to semantic concepts like *forename*, *currency* or *ISBN*. Moreover, the given unit of measurement, data format, level of abstraction etc. are derived for this attribute (if existing) and stored in the data schema. Thus, making a good job in selecting this abstract data type is very important not only because of an appropriate description of the actual representation form, but it also enables an association of alternative representation forms to this attribute and thus accomplishes a more realistic data pollution process.
- A (dis)similarity/distance measure per attribute which is able to compare two values of the attribute's data type in a (semantically) meaningful way.
- Integrity constraints such as UNIQUE, NOT NULL or functional dependencies.

Recall, the derived data schema only serves as a proposal which can be reviewed and revised by the user at the end of this step.

In the second step, sets of representation and error schemas are derived from the data schema. This reasoning is accomplished by algorithms that determine which attributes are associated with which alternative representation forms and error classes because not every representation form and error class is suitable for each attribute. As in the case of the data schema, the resultant representation and error schemas are considered as proposals and can be reviewed as well as revised by the user at the end of this step.

Before the actual pollution process can be started, the schema-independent parameters need to be configured, too. Because the settings of these parameters depend on the intended use of the generated data set, they cannot be automatically derived from the input data, but need to be set by the user herself. The most important schema-independent parameters are:

- The duplicate distribution defines how many duplicate clusters of which size have to be generated.
- The degree of pollution defines to which extent the objects of the test data set have to be polluted. In our prototype DaPo (see Section 5), we model it by the inverse of the average similarity[8] between all objects of the same duplicate cluster[9] and hence configure it by specifying a target similarity. Alternatively, a fixed number of iterations can be defined instead of a desired degree of pollution.
- Optionally, the number of simulated sources can be explicitly set by the user.

Since some of these parameters (e.g. number of sources and degree of pollution) can be helpful to reason appropriate representation and error schemas, they can also be used as input to the second step of the semiautomatic configuration phase.

4.2.3 Execution of the Pollution Process

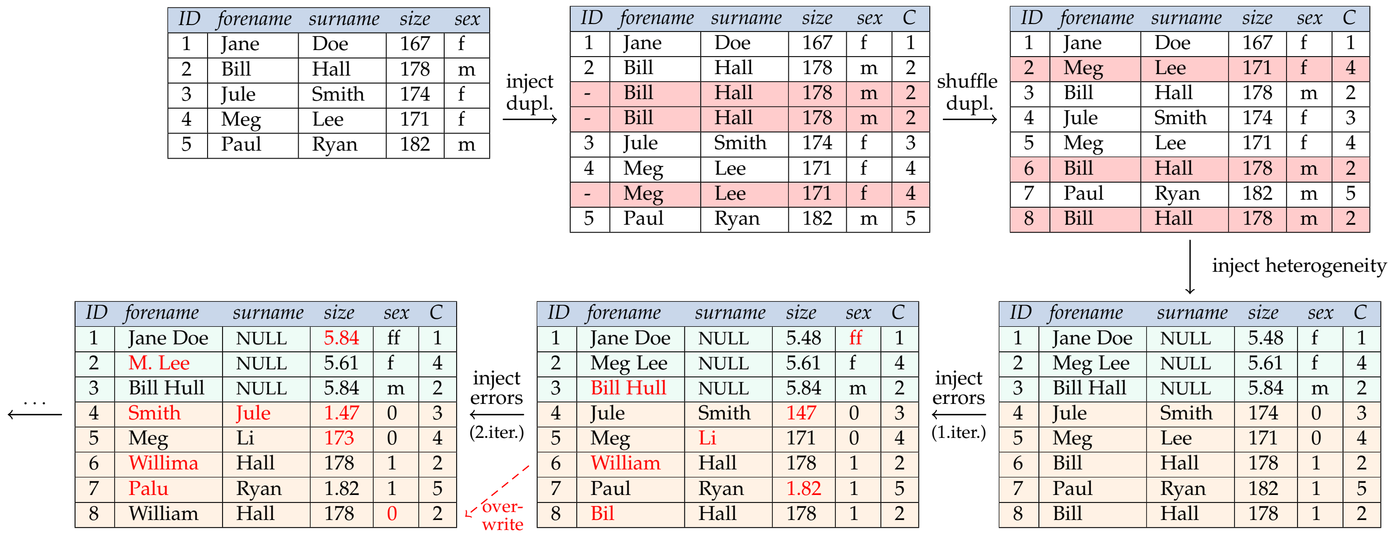
In this main phase, the data pollution process and thus the actual test data generation is executed. This execution is performed in four steps:

- **Duplicate Injection:** Following the given configuration of the duplicate distribution parameter, for each duplicate cluster C one of the input objects is selected randomly and duplicated $|C| - 1$ times where $|C|$ represents the cluster's size. Thereafter, the generated duplicates are added to the input set and we store the gold standard by assigning a (duplicate) cluster-id to each object (see also Figure 1). Finally, to distribute the newly inserted duplicates across the whole data set, this set is shuffled, but the order of the original data objects is maintained.
- **Heterogeneity Injection:** After duplicates are injected, every data object is transformed according to the representation schema of its source group. These transformations include changes in languages, data formats, encodings and units of measurements (see Section 4.1.5).
- **Error Injection:** After the source-specific transformations have been completed, the transformed data objects are polluted by injecting errors. For this purpose, the previously defined error schemas are iteratively applied to the data set until the desired degree of pollution is reached. To validate the stop condition, the current degree of pollution is computed at the beginning of each iteration and compared with the target value specified in the configuration.
  To accomplish the concept of error inheritance, at the end of each iteration some objects are overwritten with other objects of the same duplicate clusters at random. To control whether and how often such an overwriting is applied, a probability can be defined in each error schema. This probability corresponds to an initial setting which holds for the first iteration and then decreases automatically before each of the following iterations. Of course, the representation of an overwritten object needs to be transformed if its values are copied from an object belonging to another source group. The corresponding transformation process can be derived from the representation schemas of both objects.

Example 1: To illustrate the pollution process, we consider the example depicted in Figure 5. At the beginning (top left), a clean data set with five objects (each a single tuple) is given as input. In the first step, three duplicates are injected into this data set by selecting two of the original objects (i.e. 2 and 4) at random (note that the cluster-id is stored in attribute 'C'). After injection, all eight objects are shuffled, but the order between the original five objects is maintained (i.e. only the positions of the three injected objects change). In the next step, every object

8. The similarity between two objects is defined as the weighted average Jaro-Winkler similarity between their attribute values.

9. The (average) similarity between two non-duplicates is primarily determined by the nature of the given input set and is only marginally affected by the pollution process.



ID	forename	surname	size	sex
1	Jane	Doe	167	f
2	Bill	Hall	178	m
3	Jule	Smith	174	f
4	Meg	Lee	171	f
5	Paul	Ryan	182	m

ID	forename	surname	size	sex	C
1	Jane	Doe	167	f	1
2	Bill	Hall	178	m	2
-	Bill	Hall	178	m	2
-	Bill	Hall	178	m	2
3	Jule	Smith	174	f	3
4	Meg	Lee	171	f	4
-	Meg	Lee	171	f	4
5	Paul	Ryan	182	m	5

ID	forename	surname	size	sex	C
1	Jane	Doe	167	f	1
2	Meg	Lee	171	f	4
3	Bill	Hall	178	m	2
4	Jule	Smith	174	f	3
5	Meg	Lee	171	f	4
6	Bill	Hall	178	m	2
7	Paul	Ryan	182	m	5
8	Bill	Hall	178	m	2

ID	forename	surname	size	sex	C
1	Jane Doe	NULL	5.48	f	1
2	Meg Lee	NULL	5.61	f	4
3	Bill Hall	NULL	5.84	m	2
4	Jule	Smith	174	0	3
5	Meg	Lee	171	0	4
6	Bill	Hall	178	1	2
7	Paul	Ryan	182	1	5
8	Bill	Hall	178	1	2

ID	forename	surname	size	sex	C
1	Jane Doe	NULL	5.48	ff	1
2	Meg Lee	NULL	5.61	f	4
3	Bill Hull	NULL	5.84	m	2
4	Jule	Smith	147	0	3
5	Meg	Li	171	0	4
6	William	Hall	178	1	2
7	Paul	Ryan	1.82	1	5
8	Bil	Hall	178	1	2

ID	forename	surname	size	sex	C
1	Jane Doe	NULL	5.84	ff	1
2	M. Lee	NULL	5.61	f	4
3	Bill Hull	NULL	5.84	m	2
4	Smith	Jule	1.47	0	3
5	Meg	Li	173	0	4
6	Willima	Hall	178	1	2
7	Palu	Ryan	1.82	1	5
8	William	Hall	178	0	2

Fig. 5. Example of the data pollution process with three injected duplicates (red-colored tuples) and two source groups (mint-/orange-colored tuples).

is transformed according to the representation schema of its source group. In this example, we assume two of such groups which are indicated by the mint- and orange-colored tuples. In the case of the first group, fore- and surnames are stored in a single attribute. Thus, for each of the first three objects, these two values are merged and stored as forename whereas the surname is set to NULL. Moreover, the size values are measured in feet instead of cm and hence were transformed accordingly. In the case of the second group, the gender information are encoded by the two values '0' (female) and '1' (male). The values of the remaining attributes remain unchanged. After these transformations have been completed, the first iteration of the error injection process starts and injects seven errors (red-colored values) into the objects of both groups. At the end of this iteration, the values of object 8 are overwritten by the values of object 6 (error inheritance). Thereafter, the second iteration starts and injects eight errors including a transposition of the fore- and surname of object 4. Then, the pollution process continues until the stop condition is satisfied.

5 Scalable Data Pollution with DaPo

In this section, we present our prototypical data pollution tool called DaPo which is based on the framework introduced in the previous section. In the development of DaPo, we focused on the main part (i.e. the pollution process) so far. For that reason, its current implementation contains only some rudimentary approaches for reasoning data, representation and error schemas from the data profile computed in the analysis phase.

As underlying platform for DaPo, we chose the cluster computing framework Apache Spark which is especially suitable for two reasons:

- Because it is a cluster computing framework, parallelization and distribution are two of the main aspects which are enabled and optimized by Spark.
- One of the main reasons for developing Spark is the analysis of large data sets [15]. Thus, it is especially useful for an efficient data profiling because it provides several operations for distributed statistics computation.

Besides these two aspects, Spark has several other benefits. First, the Spark API abstracts from parallelization and distribution issues so that developing and deploying a potentially distributed application is very comfortable for the user. Second, Spark does not only offer the option of deploying an application on a computer cluster, but also on a standard desktop computer which enables a user spectrum ranging from big to small companies even including private persons. Finally, because of its in-memory approach, Spark is particularly well suited for iterative processes such as the one injecting errors in DaPo.

Spark provides APIs for the programming languages Scala, Java, Python and R. Since Spark itself has been (and still is) implemented in Scala, this API is the most advanced one. For this reason, DaPo is implemented in Scala, too.

5.1 Apache Spark

Before we begin to describe in which way data pollution is parallelized and distributed by DaPo in Section 5.2, we introduce some background information on Apache Spark [15], [22].

Apache Spark is a cluster computing framework for large-scale data processing which focuses on efficiency and fault tolerance in the execution of iterative and interactive (data mining) algorithms. To enable these two properties, the high performant in-memory data structure *Resilient Distributed Datasets* (short RDDs) has been developed [22]. A large range of parallel high-level operations on this data structure as well as constructs like shared variables can be used via several APIs and ensure a comfortable programming of distributed applications.

Spark is based on a master-slave architecture where a driver process (typically running on the master node) controls the program flow and distributes tasks to several executor processes (each running on a worker node). An important aspect in distributing tasks is to consider data locality and hence to reduce the number of data exchanges between different worker nodes to a minimum ('ship code not data'). A worker node has two jobs. First, it executes the tasks which have been assigned by the driver (i.e. the executor process). Second, it provides its main memory as in-memory storage for RDDs. If a worker node fails, the application runs on the remaining worker nodes. A Spark application can be executed in local mode (single desktop computer) or distributed mode (computer cluster).

An RDD is a partitioned parallel collection of objects which can be distributed among several cluster nodes. In principle, it corresponds to an abstraction of distributed main memory which enables an efficient reusage of interim results. Logically, it is a read-only data structure which can be created by parsing data from secondary memory, parallelizing a non-distributed

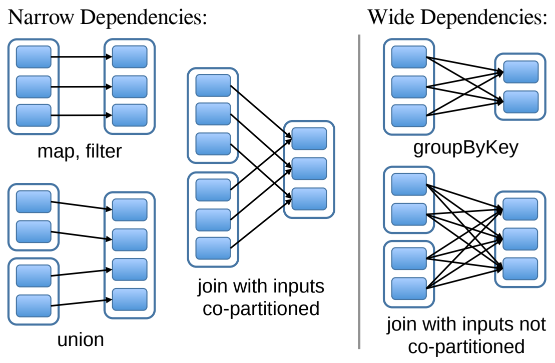

Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned

Fig. 6. Examples of narrow- and wide-dependencies [22].

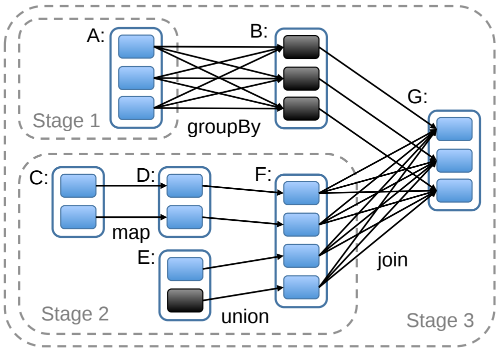


Fig. 7. Example of a lineage graph [22].

collection, or applying a transformation (e.g. `map`, `filter` or `join`) to existing RDDs.

Transformations cause dependencies between individual RDDs. These dependencies can be categorized into two classes. In the case of narrow-dependencies, for each input partition there is only one output partition that depends on it. In contrast, in the case of wide-dependencies, several output partitions can depend on the same input partition. Some transformations, such as `map`, `filter` or `union`, always cause narrow dependencies, but some transformations, such as `join`, can also cause wide-dependencies or even always implicate wide-dependencies (e.g. `groupByKey`). Examples of such dependencies are graphically illustrated in Figure 6 where RDDs are presented by frames and partitions are presented by blue-colored boxes. Whereas narrow-dependencies enable a pipelining of transformations and thus can considerably increase the degree of parallelization, wide-dependencies end in so-called *shufflings* where objects of one input partition need to be distributed among several output partitions. Since such shufflings often result in data transfers between different worker nodes, wide-dependencies usually imply a significant synchronization overhead which include serialization and several I/O accesses (disk or network). Besides transformations, Spark provides so-called *actions* which do not create new RDDs, but store their results on secondary storage (e.g. in form of txt-files) or return them directly to the user. Examples of such actions are `reduce`, `saveAsTextFile` or `count`.

The creation of RDDs is lazy which means that the objects of an RRD do not reside in physical memory until they are required as input to a subsequent processing step. This lazy evaluation is accomplished by maintaining a directed acyclic lineage graph modeling the dependencies between the individual RDDs and has two main reasons. First, it reduces runtime (no unnecessary computations) and storage requirements. Second, it increases fault tolerance because it helps to reconstruct (interim) results of failed cluster nodes. Nevertheless, if an (interim) result serves as input to multiple tasks, a lazy evaluation cause a computation overhead because this result is computed more than once. To avoid such recomputations and to decrease the effort in the case of node failures, RDDs can be explicitly persisted in memory (main or secondary). RDDs are partitioned based on so-called *partitioners* which are provided by Spark (e.g. hash or range) or can be explicitly defined by the user. The crucial aspect in data partitioning is to enable a local data processing for as many tasks as possible. An example of a lineage graph is presented in Figure 7 where blue-colored boxes represent non-persisted partitions, black-colored boxes represent persisted partitions and a stage corresponds to a pipeline of narrow-dependency operations.

5.2 Parallel and Distributed Data Processing in DaPo

An effective distribution of the DaPo workflow by using Spark depends on several issues which we discuss in this section. Note, in most processing steps of DaPo, data values are processed in an object-based fashion. Therefore, data is partitioned horizontally (i.e. all values of a data object belong to the same partition) instead of vertically (i.e. all values of an attribute belong to the same partition).

5.2.1 Data Profiling

The Spark APIs provide several operations for statistical data analysis such as `countByValue()`, `stats()` and `count()`. In DaPo, we use some of these operations to construct the attribute-based data profile during the first main phase.

5.2.2 Duplicate & Error Injection

The crucial aspect for an efficient parallelization and distribution of data operations in Spark is whether such an operation only concerns data belonging to the same partition (narrow-dependencies) or it also concerns data belonging to different partitions (wide-dependencies)

In the process of duplicate injection, each duplicate cluster is created by selecting a single data object and by replicating it several times. The replication process can be done for each duplicate cluster independently (narrow-dependency) when the driver specifies how many duplicate clusters of which sizes have to be created by the individual worker nodes.

In the process of error injection, we have to distinguish between row-, column- and field-errors. Whereas row- and field-errors are limited to single data objects and hence only concern data of the same partition (narrow-dependencies), column-errors process values across multiple data objects and hence can concern data which is stored at different partitions (wide-dependencies).

Whether the process of error inheritance is a narrow- or wide-dependency operation depends on whether all objects of a duplicate cluster are always stored in the same partition or can also be stored in different partitions (see next section).

5.2.3 Data Partitioning

A common method to increase the performance of distributed applications is to avoid data shuffling by improving data locality. Some operations in the DaPo workflow process the objects of a duplicate cluster together. These operations include:

- injection of duplicates
- computation of the degree of pollution at the beginning of each iteration
- error inheritance

Since these objects are processed together, it is beneficial to store them within the same partition (and hence on the same worker

TABLE 2
Data Sets Used as Experimental Input

Name	#Objects	Size (MB)
musicbrainz-10k.csv	10,000	0.89
musicbrainz-100k.csv	100,000	8.95
musicbrainz-1mio.csv	1,000,000	89.71
musicbrainz-10mio.csv	10,000,000	896.75
musicbrainz-100mio.csv	100,000,000	8,552.11

TABLE 3
Computer Cluster Configurations Used in the Experiments

Name	Mode	HDFS	#Nodes	Spark-Executors		
				quantity	#cores	RAM
local	local	✗	1	1	4[1]	5 GB
cluster-1	distrib.	✓	2	1	4	6 GB
cluster-2	distrib.	✓	3	2	8	12 GB
cluster-4	distrib.	✓	5	4	16	24 GB
cluster-8	distrib.	✓	9	8	32	48 GB

[1] The four cores were shared with the driver process.

node). For that reason, all data objects are partitioned based on their cluster-id throughout the whole pollution process.

Of course, other partitioning criteria, such as object groups, can be beneficial as well. However, if object groups are used to simulate multiple data sources in an integration scenario and most of the generated duplicates have to be inter-source duplicates, (almost) all objects of a duplicate cluster belong to different groups and a partitioning based on object groups as well as duplicate clusters is not possible. Nevertheless, although all objects of the same group are polluted based on the same error schema, each of these objects is polluted independently in the case of row- and field-errors. Thus, as in the general discussion, data locality is only beneficial in the application of column-errors which are typically less often applied to data objects than row- or field-errors. Moreover, object representations are always transformed independently (heterogeneity injection). Therefore, a partitioning based on cluster-ids should be preferred to a partitioning based on object groups if these two criteria are in conflict.

5.2.4 Data Caching

In DaPo, in every iteration of the error injection process, the degree of pollution is computed. Each of these computations causes a branch in the lineage graph and hence would be executed based on the original input values if the interim results are not explicitly persisted in main memory (or disk if necessary). Obviously, such a caching increases the storage requirements, but decreases runtime to a large extent. Moreover, in DaPo such an explicit caching is even essential, because errors are injected based on probabilities. Thus, the pollution process is non-deterministic and its result cannot be simply reproduced by re-executing the underlying code (i.e. different executions lead to different results even if the same error schemas are used). As a consequence, without caching, the degree of pollution would always be computed based on a data set which is different to the one produced by the latest iteration of the injection process. To decrease storage requirements, interim results are unpersisted if not required anymore.

6 Experimental Evaluation

In this section, we present a set of experiments that were conducted to evaluate the efficiency and scalability of DaPo with respect to a varying size of the input set, used cluster configuration or desired degree of pollution.

6.1 Input Data Sets

To evaluate the efficiency and scalability of DaPo in the generation of large test data sets, we require large input data sets. For this purpose, we used the free available MusicBrainz database[10] containing 11 GB meta data from the music domain (interpret, songs, recordings, etc.) stored in a complex relational schema and extracted a single CSV-file of 1.4 GB with around 16 million rows (each a music track) and 8 columns. Based on this file, we generated the five input data sets depicted in Table 2.

6.2 Experimental Set Ups

To run the experiments on a computer cluster, we connected nine standard desktop computers (Dell Optiplex 980) each with four 2.67 GHz CPU cores (Intel Core i5-750), 8 GB RAM and a 500 GB SATA-II hard drive with 7200 rpm in a switched star topology with full duplex using a 1 Gbit/s Switch. On each cluster node ran the same software including Ubuntu Server 12.04, Apache Spark 1.5.1 and Apache Hadoop 2.6.0. To decrease configuration and performance overhead, we did not use virtual layers and ran Spark in standalone mode. From Hadoop only the distributed file system HDFS was used and configured in such a way that all data objects were available on each node (full redundancy). One of the nine nodes was configured as Spark master and HDFS-NameNode. The other eight nodes were configured as Spark worker and HDFS-DataNodes.

To evaluate scalability, we used different configurations of this computer cluster (see Table 3). In four of these configurations, we deployed DaPo in distributed mode while varying the number of available worker nodes. In each case, the driver ran on the master node (allocating 2 GB RAM) and every executor ran on a separate worker node (allocating 6 GB RAM). As persistence layer, we used HDFS. In the fifth configuration, we deployed DaPo in local mode and hence on a single cluster node. Thus, the driver (2 GB RAM) as well as the only executor (5 GB RAM) ran on the same computer sharing the given resources. As persistence layer, we used the local file system.

We observed that the duplicate distribution influences the runtime of the generation process only marginally. Therefore, the relative amount of duplicates was set to 0.1, the maximal size of a duplicate cluster was set to 20 and the distribution of the duplicate cluster sizes was computed based on the equal-pair strategy (i.e. the total number of duplicate pairs is equal for each cluster size) in each experiment.

To minimize the influence of external factors on the experimental results, we performed each of the conducted experiments five times and used the average runtimes as final results.

Conceptually, heterogeneity injection is nothing else than an additional iteration of the error injection process. Therefore, we skipped this step by defining a single source group.

6.3 Experimental Results

In the first experiment, we varied the size of the input data set (and hence of the generated test data set) and evaluated the runtime of DaPo for all five cluster configurations. The target similarity was fixed to 0.85 which results in two iterations of the error injection process[11]. The results of this experiment are presented in Figure 8 and demonstrate that DaPo was able to

10. https://musicbrainz.org/doc/MusicBrainz_Database

11. This similarity setting is adopted from the real-life data set CDDB.

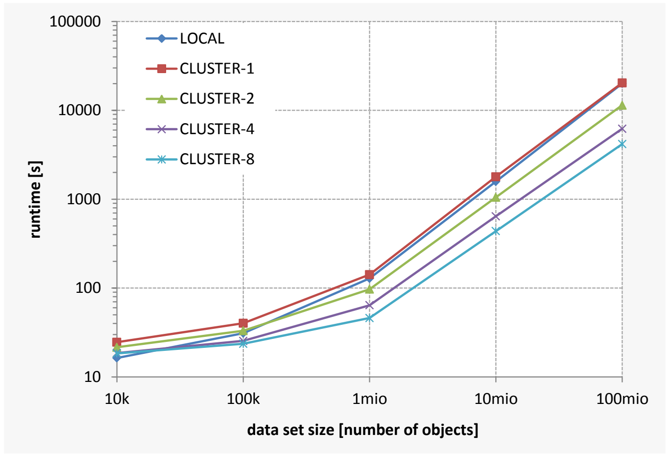


Fig. 8. Runtime behaviors of the individual cluster configurations for input data sets of different sizes (note that both axes scale logarithmically).

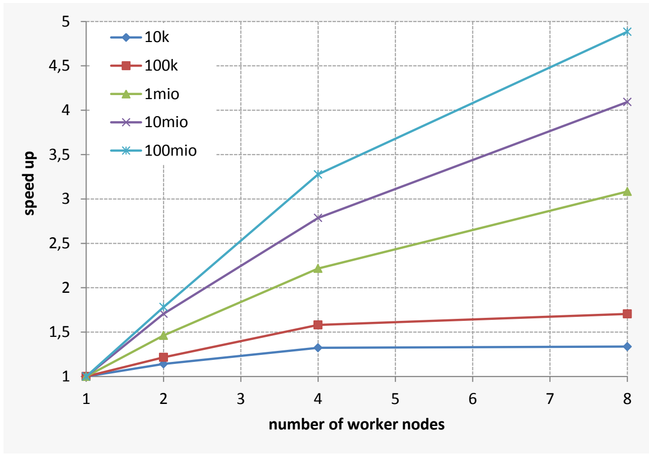


Fig. 9. Speed up resulting from increasing the number worker nodes for different sizes of the input data set.

process all these data sets easily if enough hardware resources were given. As expected, runtime increased with a growing size of the input data set for each cluster configuration. Up to a size of one million objects, the increase of runtime was sub-linear, but for larger data sets this increase became super-linear. By comparing the results of the individual cluster configurations, we can see that the use of a computer cluster did not have much positive impact on runtime when processing one of the smaller data sets. On the contrary, DaPo achieved the shortest runtimes for the smallest data set when it was deployed in local mode. Nevertheless, for data sets with more than one million objects, adding additional worker nodes reduced runtime considerably. This runtime behavior is well illustrated in Figure 9 where the speed up resulting from increasing the number of worker nodes is presented in regard with the size of the input set. In general, the larger the input set, the greater became the speed up and hence the more beneficial was the addition of extra worker nodes. Logically, the achieved speed up became better if the relative amount of the constant orchestration overhead became smaller and hence if the required amount of work (i.e. number of processed objects) per worker node became larger. This circumstance is perfectly illustrated in Figure 9 where the speed up increased more (slope of the presented lines) when the size of the input data set increased (follow the different lines of a specific section from bottom to top) or the number of worker nodes decreased (follow the sections of a specific line from right to left).

In the same experiment, we studied the runtime behavior of the individual (sub)phases of DaPo. The most important of them are presented in Figure 10. The four main phases are blue-colored and include loading the input set (import), analyzing the input set and reasoning error schemas (preparation)[12], polluting the data objects and writing the generated test data set to disk (export). In addition to these main phases, we also considered the subphases of the pollution phase. These phases are green-colored and include injecting exact duplicates, injecting errors, calculating the average duplicate similarity and distributing objects of the same duplicate cluster across the whole data set (shuffle duplicates). Note that the presented runtimes of the error injection phase do not include the runtimes of the similarity calculation phase although the latter is executed at the beginning of each iteration of the first.

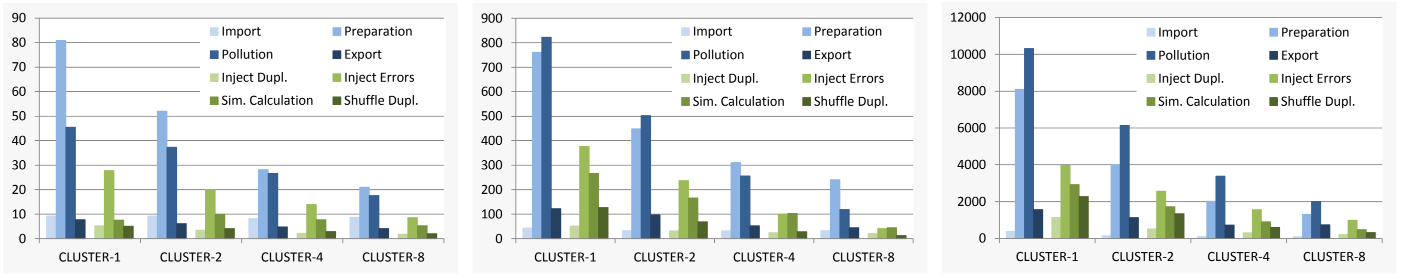
12. Since we used a rudimentary approach for schema reasoning, almost all of the entire runtime of these two phases results from the analysis part and we consider them as a single phase in this evaluation.

Figure 10 presents the runtimes (in seconds) of these phases for one, ten and hundred million objects while considering different numbers of worker nodes. As expected, the preparation and pollution phases dominated the import and export phases by far. Interestingly, the preparation time was much longer than the pollution time for small data sets, but the contrary was the case for large data sets. Nevertheless, both phases scaled very well for each of the three input sets so that an addition of extra worker nodes was always valuable. The most dominating parts of the pollution process were the error injection and the similarity calculation phases where the latter one scaled worse than the former so that the relative amount of time spent for calculating similarity increased with a growing number of worker nodes. However, in the case of the larger data sets, the similarity calculation time also decreased significantly when we add additional worker nodes. Recall that most of existing data generation tools do not provide the setting of a target similarity and hence do not need to spend time in such calculations.

In the second experiment, we varied the size of the target similarity (and hence the number of required iterations) and evaluated the runtime of DaPo for all five cluster configurations. The size of the input data set was fixed to one million objects. The results of this experiment are presented in Figure 11. As one can see, runtime increased disproportionately with a decreasing target similarity. This can be explained as follows: For most similarity measures, the similarity between two (almost) identical objects sinks rapidly by injecting a few errors, but the similarity between two (already) dissimilar objects sinks only noticeably if many (additional) errors are injected. As a consequence, the pollution effect of DaPo sank with every additional iteration (even if the same number of errors was injected) and the number of required iterations (and hence runtime) increased disproportionately with a decreasing target similarity. Note, the execution time per iteration was stable (i.e. every iteration required (almost) the same runtime). Noticeable, runtime increased dramatically with a decreasing target similarity if DaPo was deployed in local mode. This can be explained by the limited resources of the single cluster node.

6.4 Experimental Conclusions

The performed experiments proved that DaPo is able to generate test data sets with ten million objects on a single desktop machine and hundred million objects on a computer cluster with eight worker nodes in acceptable runtimes. Thus, for smaller data sets, DaPo is also useful if no computer cluster is available. DaPo in general and each of its computation phases



(a) 1 million objects (b) 10 million objects (c) 100 million objects

Fig. 10. Runtime behavior (in seconds) of the individual (sub)phases for different cluster configurations and input data sets.

in particular scaled very well even if the achieved speed up was always sub-linear and shrank with a decreasing size of the input set and/or an increasing number of worker nodes. Noticeable, the preparation as well as the similarity calculation phase required a significant amount of the total runtime. Thus, the additional features of an automatic preconfiguration and providing a parameter for setting the desired degree of pollution came with considerable cost.

The runtime of DaPo was almost proportional to the number of required iterations. Since its memory requirements primarily depend on the size of the input data set and this size does not change during the error injection process, the number of iterations can theoretically be increased at pleasure.

6.5 Comparison with Existing Tools

In Section 3, we presented five tools that can be used to generate test data sets for duplicate detection. With respect to efficiency, we compare DaPo with these systems in two ways. First by deploying it in local mode and second by deploying it in distributed mode with eight worker nodes (see Table 1). In the first case, DaPo required 31 seconds, 2.1 minutes and 26 minutes to process one hundred thousand, one million and ten million objects and hence was already faster than each of these tools except DBGen (recall that TDGen and ProbGee were even not able to process one million objects). In the second case, DaPo required 24 seconds (100k objects), 46 seconds (1mio objects) and 7.3 minutes (10mio objects) and thus came closer to the runtime of DBGen even if this tool was still faster. Since none of these tools provides an automatic preconfiguration, we think that the comparison is fairer if we omit the expensive analysis phase of DaPo by using predefined error schemas. The corresponding runtimes are added to Table 1 (marked with an *) and are another step closer to the runtimes of DBGen.

Unlike some of these tools, DaPo currently do not provide a GUI and can only be used via command line. However, DaPo is the only tool that scales horizontally and thus able to exploit the computation power of a computer cluster. Compared to ProbGee, TDGen and GeCo, DaPo achieved very much shorter runtimes and provides similar (or even more) configuration options to the user. DBGen is significantly faster than DaPo, but it is very inflexible (not schema-independent, less configuration options) and generates test data being much less realistic because it is a synthesization tool. FebrlDG achieved similar runtimes than DaPo, but has the same (or at least similar) shortcomings as DBGen. Moreover, it was not able to process 10 million objects because of heap space issues. In conclusion, DaPo is superior to these tools in almost all important aspects.

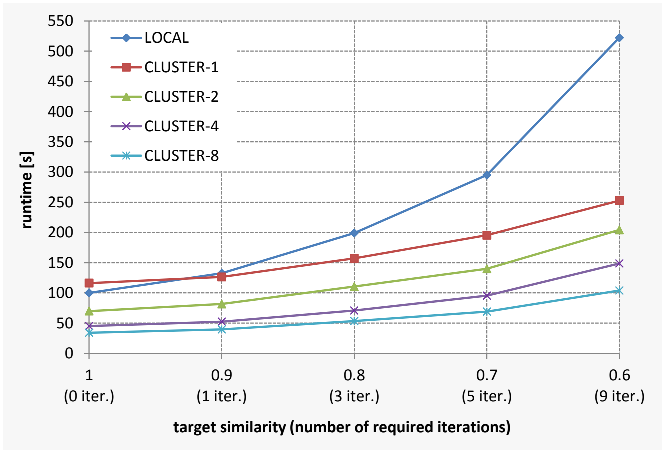


Fig. 11. Runtime behaviors of the individual cluster configurations for different values of the target similarity.

7 Conclusion & Future Work

Duplicate detection is an important challenge in today's data management where error-prone and heterogenously modeled data objects are provided by a large number of autonomic data sources. To evaluate the quality of new duplicate detection algorithms or to adjust an existing algorithm to a specific application scenario, realistic test data sets are required. Of course, in times of big data, algorithms need to be efficient and scalable. As a consequence, test data sets need to be large. Existing test data generation tools, however, either lack in efficiency (i.e. large test data sets cannot be generated in acceptable runtimes), quality (i.e. the generated test data sets are not very realistic) or flexibility (i.e. data generation is limited to a predefined schema/domain). In this paper, we proposed an efficient, scalable and domain-independent framework for generating large and realistic test data sets for duplicate detection and presented a prototypical implementation of this framework called DaPo which is based on Apache Spark.

In Section 2, we elaborated a list of desired properties which should be provided by such a test data generation tool. In DaPo, these desiderata are accomplished as follows:

- **Efficiency & Scalability:** As demonstrated by our experimental evaluations, DaPo is very efficient and scales vertically as well as horizontally. The last two desiderata are realized by using the cluster computing framework Apache Spark as underlying platform.
- **Schema Independence:** DaPo is schema- and domain-independent by nature because it is a data pollution tool which gets an arbitrary data source as input. This independence is additionally supported by the first two main phases which derive an enriched data schema from the input source automatically.
- **Realistic Data Values & Patterns:** As demonstrated by the tool DBGen [7], the generation of realistic data values and patterns is difficult if these values are synthesized.

Therefore, we accomplish this desiderata by designing DaPo as a data pollution tool and hence by using a real-life data set as basis.

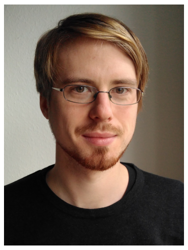
- **Realistic & Variable Error Patterns:** This desiderata is met by introducing the concepts of source/object groups (to simulate multiple data sources), representation/error schemas and error inheritance as well as by providing a large range of error classes.
- **Simple but Adaptable Configuration:** On one hand, DaPo provides a large set of parameters (including a desired degree of pollution) which enable the user to adjust the tool to a particular application scenario. On the other hand, the mechanisms for computing an input-specific preconfiguration of these parameters automatically help the user by reducing the configuration effort to a large extent.

Our data pollution framework presented in Section 4 is broadly defined in a generic way. Nevertheless, there are some aspects being worth for further considerations. The current distinction of error classes into row-, column- and field-errors results from considering single relational tables. Relational databases, however, often consist of multiple tables which are related by means of foreign keys. Moreover, the input data set can also be provided in a non-relational form, such as key-value and wide column stores, XML-files or JSON-documents [23], [24], so that additional categories of error classes (e.g. for object references and nested or schema-less objects) are required.

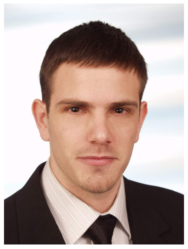
The development of DaPo is still in progress and there are several aspects which require further considerations. The most important of them concerns the development of more sophisticated approaches for reasoning data, representation and error schemas. Moreover, the current implementation of DaPo does not provide any mechanisms to extract complex integrity constraints from the input source. Finally, it does not have a GUI and its input as well as output are limited to CSV-files.

References

[1] A. Doan, A. Halevy, and Z. G. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012.
[2] E. Rahm and H. H. Do, “Data cleaning: Problems and current approaches,” *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, 2000.
[3] V. Ganti and A. D. Sarma, *Data Cleaning: A Practical Perspective*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.
[4] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, “Duplicate Record Detection: A Survey,” *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 1–16, 2007.
[5] F. Naumann and M. Herschel, *An Introduction to Duplicate Detection*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
[6] P. Christen, *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*, ser. Data-Centric Systems and Applications. Springer, 2012.
[7] M. Hernández and S. Stolfo, “Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem,” *Data Min. Knowl. Discov.*, vol. 2, no. 1, pp. 9–37, 1998.
[8] D. Menestrina, S. Whang, and H. Garcia-Molina, “Evaluating Entity Resolution Results,” *PVLDB*, vol. 3, no. 1, pp. 208–219, 2010.
[9] X. L. Dong and D. Srivastava, *Big Data Integration*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
[10] L. Getoor and A. Machanavajjhala, “Entity resolution for big data,” in *KDD*, 2013, p. 1527.
[11] P. Christen, “Development and user experiences of an open source data cleaning, deduplication and record linkage system,” *SIGKDD Explorations*, vol. 11, no. 1, pp. 39–48, 2009.
[12] P. Christen and D. Vatsalan, “Flexible and extensible generation and corruption of personal data,” in *CIKM*, 2013, pp. 1165–1168.
[13] T. Bachteler and J. Reiher, “Tdgen: A test data generator for evaluating record linkage methods,” German Record Linkage Center, Tech. Rep. wp-grlc-2012-01, 2012.
[14] S. Friedrich and W. Wingerath, “Evaluation of tuple matching methods on generated probabilistic data,” Master’s thesis, University of Hamburg, 2012.
[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in *HotCloud*, 2010.
[16] J. Verhoeff, “Error detecting decimal codes, vol. 29, math,” *Centre Tracts. Math. Centrum Amsterdam*, 1969.
[17] K. Church and W. Gale, “Probability scoring for spelling correction,” *Statistics and Computing*, vol. 1, no. 2, pp. 93–103, 1991.
[18] P. Christen and A. Pudjijono, “Accurate Synthetic Generation of Realistic Personal Information,” in *PAKDD*, 2009, pp. 507–514.
[19] X. Dong, L. Berti-Equille, Y. Hu, and D. Srivastava, “Global detection of complex copying relationships between sources,” *PVLDB*, vol. 3, no. 1, pp. 1358–1369, 2010.
[20] K. Tran, D. Vatsalan, and P. Christen, “GeCo: an online personal data generator and corruptor,” in *CIKM*, 2013, pp. 2473–2476.
[21] Z. Abedjan, L. Golab, and F. Naumann, “Profiling relational data: a survey,” *VLDB J.*, vol. 24, no. 4, pp. 557–581, 2015.
[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” in *NSDI*, 2012, pp. 15–28.
[23] V. N. Gudivada, D. Rao, and V. V. Raghavan, “NoSQL Systems for Big Data Management,” in *SERVICES*, 2014, pp. 190–197.
[24] F. Gessert and N. Ritter, “Scalable data management: NoSQL data stores in research and practice,” in *ICDE*, 2016, pp. 1420–1423.



Kai Hildebrandt is an alumni of the University of Hamburg. He received his MSc. from the University of Hamburg in 2015. His research interests include big data processing, duplicate detection and the impact of big data on privacy and society.



Fabian Panse is a postdoctoral teacher and researcher at the University of Hamburg where he is part of the databases and information systems group. He received his Ph.D. from the University of Hamburg in 2014. His research interests include big data processing, data quality, information integration (especially duplicate detection) and uncertain databases. He has held workshop and conference talks on his published work on several occasions.

Niklas Wilcke is an alumni of the University of Hamburg. He received his MSc. from the University of Hamburg in 2015. His research interests include big data processing, duplicate detection and distributed systems. His Master’s thesis addressed the topic of a scalable distributed duplicate detection framework called SDDF.

Norbert Ritter is a full professor of computer science at the University of Hamburg, where he heads the databases and information systems group. He received his Ph.D. from the University of Kaiserslautern in 1997. His research interests include distributed and federated database systems, transaction processing, caching, cloud data management, information integration and autonomous database systems. He has been teaching NoSQL topics in various database courses for several years.