BigDansing: A System for Big Data Cleansing

Zuhair Khayyat[†*] Ihab F. Ilyas[‡*] Alekh Jindal[♯] Samuel Madden[♯]
Mourad Ouzzani[§] Paolo Papotti[§] Jorge-Arnulfo Quiané-Ruiz[§] Nan Tang[§] Si Yin[§]
[§]Qatar Computing Research Institute [♯]CSAIL, MIT
[†]King Abdullah University of Science and Technology (KAUST) [‡]University of Waterloo
zuhair.khayyat@kaust.edu.sa ilyas@uwaterloo.ca {alekh,madden}@csail.mit.edu
{mouzzani,ppapotti,jquianeruiz,ntang,siyin}@qf.org.qa

ABSTRACT

Data cleansing approaches have usually focused on detecting and fixing errors with little attention to scaling to big datasets. This presents a serious impediment since data cleansing often involves costly computations such as enumerating pairs of tuples, handling inequality joins, and dealing with user-defined functions. In this paper, we present BigDansing, a Big Data Cleansing system to tackle efficiency, scalability, and ease-of-use issues in data cleansing. The system can run on top of most common general purpose data processing platforms, ranging from DBMSs to MapReduce-like frameworks. A user-friendly programming interface allows users to express data quality rules both declaratively and procedurally, with no requirement of being aware of the underlying distributed platform. BigDansing takes these rules into a series of transformations that enable distributed computations and several optimizations, such as shared scans and specialized joins operators. Experimental results on both synthetic and real datasets show that BigDansing outperforms existing baseline systems up to more than two orders of magnitude without sacrificing the quality provided by the repair algorithms.

1. INTRODUCTION

Data quality is a major concern in terms of its scale as more than 25% of critical data in the world's top companies is flawed [31]. In fact, cleansing dirty data is a critical challenge that has been studied for decades [10]. However, data cleansing is hard, since data errors arise in different forms, such as typos, duplicates, non compliance with business rules, outdated data, and missing values.

Example 1: Table 1 shows a sample tax data D in which each record represents an individual's information. Suppose that the following three data quality rules need to hold on D: ($r1$) a zipcode uniquely determines a city; ($r2$) given two distinct individuals, the one earning a lower salary should have a lower tax rate; and ($r3$) two tuples refer to the same individual if they have similar names, and their cities are inside the same county. We define these rules as follows:

	name	zipcode	city	state	salary	rate
t_1	Annie	10001	NY	NY	24000	15
t_2	Laure	90210	LA	CA	25000	10
t_3	John	60601	CH	IL	40000	25
t_4	Mark	90210	SF	CA	88000	28
t_5	Robert	60827	CH	IL	15000	15
t_6	Mary	90210	LA	CA	81000	28

Table 1: Dataset D with tax data records

($r1$) $\phi_F : D(\mathsf{zipcode} \rightarrow \mathsf{city})$
($r2$) $\phi_D : \forall t_1, t_2 \in D, \neg(t_1.\mathsf{rate} > t_2.\mathsf{rate} \wedge t_1.\mathsf{salary} < t_2.\mathsf{salary})$
($r3$) $\phi_U : \forall t_1, t_2 \in D, \neg(\mathsf{simF}(t_1.\mathsf{name}, t_2.\mathsf{name}) \wedge \mathsf{getCounty}(t_1.city) = \mathsf{getCounty}(t_2.city))$

ϕ_F, ϕ_D, and ϕ_U can respectively be expressed as a functional dependency (FD), a denial constraint (DC), and a user-defined function (UDF) by using a procedural language. ϕ_U requires an ad-hoc similarity function and access to a mapping table to get the county information. Tuples t_2 and t_4 form an error *w.r.t.* ϕ_F, so do t_4 and t_6, because they have the same zipcode but different city values. Tuples t_1 and t_2 violate ϕ_D, because t_1 has a lower salary than t_2 but pays a higher tax; so do t_2 and t_5. Rule ϕ_U does not have data errors, *i.e.,* no duplicates in D *w.r.t.* ϕ_U. □

Cleansing the above dataset typically consists of three steps: (1) specifying quality rules; (2) detecting data errors *w.r.t.* the specified rules; and (3) repairing detected errors. Generally speaking, after step (1), the data cleansing process *iteratively* runs steps (2) and (3) until obtaining an instance of the data (*a.k.a.* a repair) that satisfies the specified rules. These three steps have been extensively studied in single-node settings [6, 7, 11, 16–18, 23, 34]. However, the main focus has been on how to more effectively detect and repair errors, without addressing the issue of scalability.

Dealing with large datasets faces two main issues. First, none of the existing systems can scale to large datasets for the rules in the example. One main reason is that detecting errors, *e.g.,* with ϕ_F and ϕ_D, or duplicates, *e.g.,* with ϕ_U, is a combinatorial problem that quickly becomes very expensive with large sizes. More specifically, if $|D|$ is the number of tuples in a dataset D and n is the number of tuples a given rule is defined on (*e.g.,* $n = 2$ for the above rules), the time complexity of detecting errors is $O(|D|^n)$. This high complexity leads to intractable computations over large datasets, limiting the applicability of data cleansing systems. For instance, memory-based approaches [6, 7] report performance numbers up to 100K tuples while a disk-based approach [17] reports numbers up to 1M records only. Apart from the scale

*Partially done while at QCRI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2747646.

problem, implementing such rules in a distributed processing platform requires expertise both in the understanding of the quality rules and in the tuning of their implementation over this platform. Some research efforts have targeted the usability of a data cleansing system (*e.g.,* NADEEF [7]), but at the expense of performance and scalability.

Therefore, designing a distributed data cleansing system to deal with big data faces two main challenges:

(1) *Scalability.* Quality rules are varied and complex: they might compare multiple tuples, use similarity comparisons, or contain inequality comparisons. Detecting errors in a distributed setting may lead to shuffling large amounts of data through the network, with negative effects on the performance. Moreover, existing repair algorithms were not designed for distributed settings. For example, they often represent detected errors as a graph, where each node represents a data value and an edge indicates a data error. Devising algorithms on such a graph in distributed settings is not well addressed in the literature.

(2) *Abstraction.* In addition to supporting traditional quality rules (*e.g.,* ϕ_F and ϕ_D), users also need the flexibility to define rules using procedural programs (UDFs), such as ϕ_U. However, effective parallelization is hard to achieve with UDFs, since they are handled as black boxes. To enable scalability, finer granular constructs for specifying the rules are needed. An abstraction for this specification is challenging as it would normally come at the expense of generality of rule specification.

To address these challenges, we present BIGDANSING, a Big Data Cleansing system that is easy-to-use and highly scalable. It provides a rule specification abstraction that consists of five operators that allow users to specify data flows for error detection, which was not possible before. The internal rewriting of the rules enables effective optimizations. In summary, we make the following contributions:

(1) BIGDANSING abstracts the rule specification process into a logical plan composed of five operators, which it optimizes for efficient execution. Users focus on the logic of their rules, rather than on the details of efficiently implementing them on top of a parallel data processing platform. More importantly, this abstraction allows to detect errors and find possible fixes *w.r.t.* a large variety of rules that cannot be expressed in declarative languages (Section 3).

(2) BIGDANSING abstraction enables a number of optimizations. We present techniques to translate a logical plan into an optimized physical plan, including: (i) removing data redundancy in input rules and reducing the number of operator calls; (ii) specialized operators to speed up the cleansing process; and (iii) a new join algorithm, based on partitioning and sorting data, to efficiently perform distributed inequality self joins (which are common in quality rules);

(3) We present two approaches to implement existing repair algorithms in distributed settings. First, we show how to run a centralized data repair algorithm in parallel, without changing the algorithm (Section 5.1). Second, we design a distributed version of the seminal *equivalence class* algorithm [5] (Section 5.2).

(4) We use both real-world and synthetic datasets to extensively validate our system. BIGDANSING outperforms baseline systems by up to more than two orders of magnitude. The results also show that BIGDANSING is more scalable than baseline systems without sacrificing the quality provided by the repair algorithms (Section 6).

2. FUNDAMENTALS AND AN OVERVIEW

We discuss the data cleansing semantics expressed by BIGDANSING and then give an overview of the system.

2.1 Data Cleansing Semantics

In BIGDANSING, the input data is defined as a set of data units, where each *data unit* is the smallest unit of input datasets. Each unit can have multiple associated *elements* that are identified by model-specific functions. For example, tuples are the data units for the relational model and attributes identify their elements, while triples are the data units for RDF data (see Appendix C). BIGDANSING provides a set of parsers for producing such data units and elements from input datasets.

BIGDANSING adopts UDFs as the basis to define quality rules. Each rule has two fundamental abstract functions, namely Detect and GenFix. Detect takes one or multiple data units as input and outputs a *violation*, *i.e.,* elements in the input units that together are considered as erroneous *w.r.t.* the rule:

$$\mathsf{Detect}(\mathsf{data\_units}) \rightarrow \mathsf{violation}$$

GenFix takes a violation as input and computes alternative, possible updates to resolve this violation:

$$\mathsf{GenFix}(\mathsf{violation}) \rightarrow \mathsf{possible\_fixes}$$

The language of the possible fixes is determined by the capabilities of the repair algorithm. With the supported algorithms, a *possible fix* in BIGDANSING is an expression of the form x op y, where x is an element, op is in $\{=, \neq, <, >, \geq, \leq\}$, and y is either an element or a constant. In addition, BIGDANSING has new functions to enable distributed and scalable execution of the entire cleansing process. We defer the details to Section 3.

Consider Example 1, the Detect function of ϕ_F takes two tuples (*i.e.,* two data units) as input and identifies a violation whenever the same zipcode value appears in the two tuples but with a different city. Thus, $t_2(90210, LA)$ and $t_4(90210, SF)$ are a violation. The GenFix function could enforce either t_2[city] and t_4[city] to be the same, or at least one element between t_2[zipcode] and t_4[zipcode] to be different from 90210. Rule ϕ_U is more general as it requires special processing. Detect takes two tuples as input and outputs a violation whenever it finds similar name values and obtains the same county values from a mapping table. GenFix could propose to assign the same values to both tuples so that one of them is removed in set semantics.

By using a UDF-based approach, we can support a large variety of traditional quality rules with a parser that automatically implements the abstract functions, *e.g.,* CFDs [11] and DCs [6], but also more procedural rules that are provided by the user. These latter rules can implement any detection and repair method expressible with procedural code, such as Java, as long as they implement the signatures of the two above functions, as demonstrated in systems such as NADEEF [7]. Note that one known limitation of UDF-based systems is that, when treating UDFs as black-boxes, it is hard to do static analysis, such as consistency and implication, for the given rules.

BIGDANSING targets the following *data cleansing problem*: given a dirty data D and a set of rules Σ, compute a *repair*

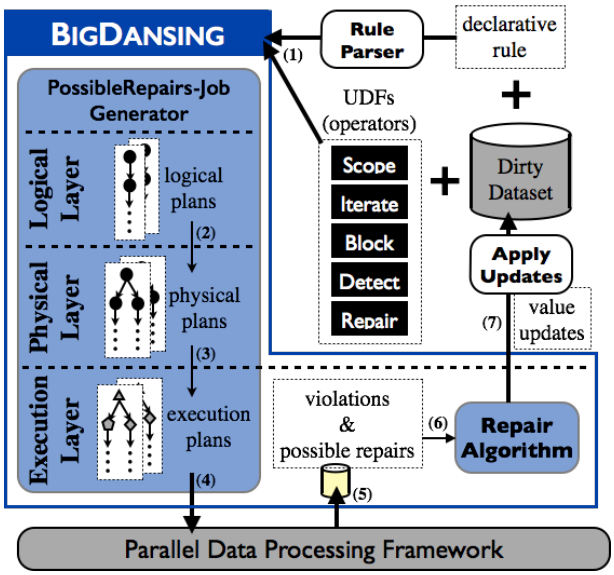


Figure 1: BigDansing architecture

which is an updated instance D' such that there are no violations, or there are only violations with no possible fixes.

Among the many possible solutions to the cleansing problem, it is common to define a notion of minimality based on the cost of a repair. A popular *cost* function [5] for relational data is the following: $\sum_{t\in D, t'\in D_r, A\in A^R} dis_A(D(t[A]), D(t'[A]))$, where t' is the fix for a specific tuple t and $dis_A(D(t[A]), D(t'[A]))$ is a distance between their values for attribute A (an exact match returns 0). This models the intuition that the higher is the sum of the distances between the original values and their fixes, the more expensive is the repair. Computing such *minimum* repairs is NP-hard, even with FDs only [5, 23]. Thus, data repair algorithms are mostly heuristics (see Section 5).

2.2 Architecture

We architected BIGDANSING as illustrated in Figure 1. BIGDANSING receives a data quality rule together with a dirty dataset from users (1) and outputs a clean dataset (7). BIGDANSING consists of two main components: the RuleEngine and the RepairAlgorithm.

The RuleEngine receives a quality rule either in a UDF-based form (a BIGDANSING *job*) or in a declarative form (a declarative rule). A job (*i.e.,* a script) defines users operations as well as the sequence in which users want to run their operations (see Appendix A). A declarative rule is written using traditional integrity constraints such as FDs and DCs. In the latter case, the RuleEngine automatically translates the declarative rule into a job to be executed on a parallel data processing framework. This job outputs the set of violations and possible fixes for each violation. The RuleEngine has three layers: the *logical*, *physical*, and *execution* layers. This architecture allows BIGDANSING to (i) support a large variety of data quality rules by abstracting the rule specification process, (ii) achieve high efficiency when cleansing datasets by performing a number of physical optimizations, and (iii) scale to big datasets by fully leveraging the scalability of existing parallel data processing frameworks. Notice that, unlike a DBMS, the RuleEngine also has an execution abstraction, which allows BIGDANSING to run on top of general purpose data processing frameworks ranging from MapReduce-like systems to databases.

(1) *Logical layer.* A major goal of BIGDANSING is to allow users to express a variety of data quality rules in a simple way. This means that users should only care about the logic of their quality rules, without worrying about how to make the code distributed. To this end, BIGDANSING provides five logical operators, namely Scope, Block, Iterate, Detect, and GenFix, to express a data quality rule: Scope defines the relevant data for the rule; Block defines the group of data units among which a violation may occur; Iterate enumerates the candidate violations; Detect determines whether a candidate violation is indeed a violation; and GenFix generates a set of possible fixes for each violation. Users define these logical operators, as well as the sequence in which BIGDANSING has to run them, in their jobs. Alternatively, users provide a declarative rule and BIGDANSING translates it into a job having these five logical operators. Notice that a job represents the logical plan of a given input quality rule.

(2) *Physical layer.* In this layer, BIGDANSING receives a logical plan and transforms it into an optimized physical plan of physical operators. Like in DBMSs, a physical plan specifies how a logical plan is implemented. For example, Block could be implemented by either hash-based or range-based methods. A physical operator in BIGDANSING also contains extra information, such as the input dataset and the degree of parallelism. Overall, BIGDANSING processes a logical plan through two main optimization steps, namely the *plan consolidation* and the *data access optimization*, through the use of specialized join and data access operators.

(3) *Execution layer.* In this layer, BIGDANSING determines how a physical plan will be actually executed on the underlying parallel data processing framework. It transforms a physical plan into an execution plan which consists of a set of system-dependent operations, *e.g.,* a Spark or MapReduce job. BIGDANSING runs the generated execution plan on the underlying system. Then, it collects all the violations and possible fixes produced by this execution. As a result, users get the benefits of parallel data processing frameworks by just providing few lines of code for the logical operators.

Once BIGDANSING has collected the set of violations and possible fixes, it proceeds to repair (*cleanse*) the input dirty dataset. At this point, the repair process is independent from the number of rules and their semantics, as the repair algorithm considers only the set of violations and their possible fixes. The way the final fixes are chosen among all the possible ones strongly depends on the repair algorithm itself. Instead of proposing a new repair algorithm, we present in Section 5 two different approaches to implement existing algorithms in a distributed setting. Correctness and termination properties of the original algorithms are preserved in our extensions. Alternatively, expert users can plug in their own repair algorithms. In the algorithms that we extend, each repair step greedily eliminates violations with possible fixes while minimizing the cost function in Section 2.1. An iterative process, *i.e.,* detection and repair, terminates if there are no more violations or there are only violations with no corresponding possible fixes. The repair step may introduce new violations on previously fixed data units and a new step may decide to again update these units. To ensure termination, the algorithm put a special variable on such units after a fixed number of iterations (which is a user

defined parameter), thus eliminating the possibility of future violations on the same data.

3. RULE SPECIFICATION

BIGDANSING abstraction consists of five logical operators: Scope, Block, Iterate, Detect, and GenFix, which are powerful enough to express a large spectrum of cleansing tasks. While Detect and GenFix are the two general operators that model the data cleansing process, Scope, Block, and Iterate enable the efficient and scalable execution of that process. Generally speaking, Scope reduces the amount of data that has to be treated, Block reduces the search space for the candidate violation generation, and Iterate efficiently traverses the reduced search space to generate all candidate violations. It is worth noting that these five operators do not model the repair process itself (*i.e.,* a repair algorithm). The system translates these operators along with a, generated or user-provided, BIGDANSING job into a *logical plan.*

3.1 Logical Operators

As mentioned earlier, BIGDANSING defines the input data through *data units* Us on which the logical operators operate. While such a fine-granular model might seem to incur a high cost as BIGDANSING calls an operator for each U, it in fact allows to apply an operator in a highly parallel fashion.

In the following, we define the five operators provided by BIGDANSING and illustrate them in Figure 2 using the dataset and rule FD ϕ_F (zipcode → city) of Example 1. Notice that the following listings are automatically generated by the system for declarative rules. Users can either modify this code or provide their own for the UDFs case.

(1) Scope removes irrelevant data units from a dataset. For each data unit U, Scope outputs a set of filtered data units, which can be an empty set.

$$\mathsf{Scope}(U) \rightarrow \mathsf{list}\langle U' \rangle$$

Notice that Scope outputs a list of Us as it allows data units to be replicated. This operator is important as it allows BIGDANSING to focus only on data units that are relevant to a given rule. For instance, in Figure 2, Scope projects on attributes zipcode and city. Listing 4 (Appendix B) shows the lines of code for Scope in rule ϕ_F.

(2) Block groups data units sharing the same blocking key. For each data unit U, Block outputs a blocking key.

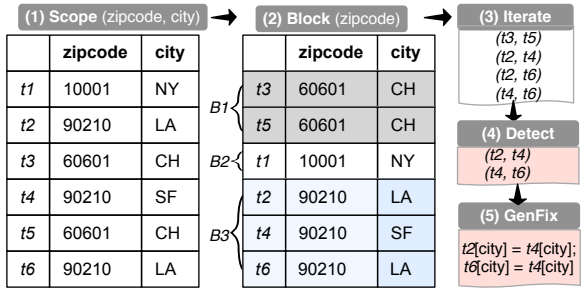
$$\mathsf{Block}(U) \rightarrow \mathsf{key}$$

The Block operator is crucial for BIGDANSING's scalability as it narrows down the number of data units on which a violation might occur. For example, in Figure 2, Block groups tuples on attribute zipcode, resulting in three blocks, with each block having a distinct zipcode. Violations might occur inside these blocks only and not across blocks. See Listing 5 (Appendix B) for the single line of code required by this operator fro rule ϕ_F.

(3) Iterate defines how to combine data units Us to generate candidate violations. This operator takes as input a list of lists of data units Us (because it might take the output of several previous operators) and outputs a single U, a pair of Us, or a list of Us.

$$\mathsf{Iterate}(\mathsf{list}\langle \mathsf{list}\langle U \rangle \rangle) \rightarrow U' \mid \langle U_i, U_j \rangle \mid \mathsf{list}\langle U'' \rangle$$

This operator allows to avoid the quadratic complexity for generating candidate violations. For instance, in Figure 2,



	zipcode	city
t1	10001	NY
t2	90210	LA
t3	60601	CH
t4	90210	SF
t5	60601	CH
t6	90210	LA

	zipcode	city
t3	60601	CH
t5	60601	CH
t1	10001	NY
t2	90210	LA
t4	90210	SF
t6	90210	LA

Figure 2: Logical operators execution for rule FD ϕ_F

Iterate passes each unique combination of two tuples inside each block, producing four pairs only (instead of 13 pairs): (t_3, t_5) from B_1, (t_2, t_4), (t_2, t_6), and (t_4, t_6) from B_3. Listing 6 (Appendix B) shows the code required by this Iterate operator for rule ϕ_F.

(4) Detect takes a single U, *a pair-U*, or a *list o Us*, as input and outputs a list of violations, possibly empty.

$$\mathsf{Detect}(U \mid \langle U_i, U_j \rangle \mid \mathsf{list}\langle U' \rangle) \rightarrow \{\mathsf{list}\langle \mathsf{violation} \rangle\}$$

Considering three types of inputs for Detect allows us to achieve better parallelization by distinguishing between different granularities of the input. For example, having 1K Us as input, rather than a single list of Us, would allow us to run 1K parallel instances of Detect (instead of a single Detect instance). In Figure 2, Detect outputs two violations, $v_1 = (t_2, t_4)$ and $v_2 = (t_6, t_4)$, as they have different values for city; and it requires the lines of code in Listing 1.

```
public ArrayList<Violation> detect(TuplePair in) {
1    ArrayList<Violation> lst = new ArrayList<Violation>();
2    if (!in.getLeft().getCellValue(1).equals(
     in.getRight().getCellValue(1))) {
3      Violation v = new Violation("zipcode => City");
4      v.addTuple(in.getLeft());
5      v.addTuple(in.getRight());
6      lst.add(v); }
7    return lst; }
```

Listing 1: Code example for the Detect operator.

(5) GenFix computes a set of possible fixes for a given violation.

$$\mathsf{GenFix}(\mathsf{violation}) \rightarrow \{\mathsf{list}\langle \mathsf{PossibleFixes} \rangle\}$$

For instance, assuming that only right-hand side values can be modified, GenFix produces one possible repair for each detected violation (Figure 2): $t_2[\mathsf{city}] = t_4[\mathsf{city}]$ and $t_6[\mathsf{city}] = t_4[\mathsf{city}]$. Listing 2 shows the code for this GenFix.

```
public ArrayList<Fix> GenFix(Violation v) {
1    ArrayList<Fix> result = new ArrayList<Fix>();
2    Tuple t1 = v.getLeft();
3    Tuple t2 = v.getRight();
4    Cell c1 = new Cell(t1.getID(),"City",t1.getCellValue(1));
5    Cell c2 = new Cell(t2.getID(),"City",t2.getCellValue(1));
6    result.add(new Fix(c1,"=",c2));
7    return result; }
```

Listing 2: Code example for the GenFix operator.

Additionally, to better specify the data flow among the different operators, we introduce a *label* to stamp a data item and track how it is being processed.

In contrast to DBMS operators, BIGDANSING's operators are UDFs, which allow users to plug in any logic. As a result, we are not restricted to a specific data model. However, for

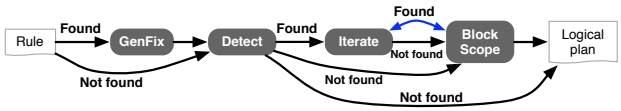


Figure 3: Planner execution flow

ease of explanations all of our examples assume relational data. We report an RDF data cleansing example in Appendix C. We also report in Appendix D the code required to write the same rule in a distributed environment, such as Spark. The proposed templates for the operators allow users to get distributed detection without any expertise on Spark, only by providing from 3 to 16 lines of Java code. The benefit of the abstraction should be apparent at this point: (i) ease-of-use for non-expert users and (ii) better scalability thanks to its abstraction.

3.2 The Planning Process

The logical layer of BigDansing takes as input a set of labeled logical operators together with a job and outputs a logical plan. The system starts by validating whether the provided job is correct by checking that all referenced logical operators are defined and at least one Detect is specified. Recall that for declarative rules, such as CFDs and DCs, users do not need to provide any job, as BigDansing automatically generates a job along with the logical operators.

After validating the provided job, BigDansing generates a logical plan (Figure 3) that: (i) *must* have at least one input dataset D_i; (ii) *may* have one or more Scope operators; (iii) *may* have one Block operator or more linked to an Iterate operator; (iv) *must* have at least one Detect operator to possibly produce a set of violations; and (v) *may* have one GenFix operator for each Detect operator to generate possible fixes for each violation.

BigDansing looks for a corresponding Iterate operator for each Detect. If Iterate is not specified, BigDansing generates one according to the input required by the Detect operator. Then, it looks for Block and Scope operators that match the input label of the Detect operator. If an Iterate operator is specified, BigDansing identifies all Block operators whose input label match the input label of the Iterate. Then, it uses the input labels of the Iterate operator to find other possible Iterate operators in reverse order. Each new detected Iterate operator is added to the plan with the Block operators related to its input labels. Once it has processed all Iterate operators, it finally looks for Scope operators.

In case the Scope or Block operators are missing, BigDansing pushes the input dataset to the next operator in the logical plan. If no GenFix operator is provided, the output of the Detect operator is written to disk. For example, assume the logical plan in Figure 4, which is generated by BigDansing when a user provides the job in Appendix A. We observe that dataset $D1$ is sent directly to a Scope and a Block operator. Similarly, dataset $D2$ is sent directly to an Iterate operator. We also observe that one can also iterate over the output of previous Iterate operators (*e.g.,* over output D_M). This flexibility allows users to express complex data quality rules, such as in the form of "bushy" plans as shown in Appendix E.

4. BUILDING PHYSICAL PLANS

When translating a logical plan, BigDansing exploits two main opportunities to derive an optimized physical plan: (i) static analysis of the logical plan and (ii) alternative

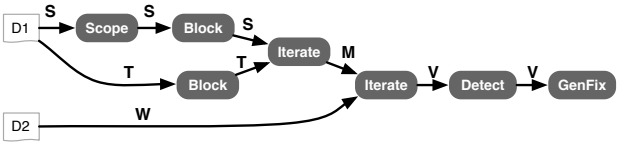


Figure 4: Example of a logical plan

translations for each logical operator. Since quality rules may involve joins over ordering comparisons, we also introduce an efficient algorithm for these cases. We discuss these aspects in this section. In addition, we discuss data storage and access issues in Appendix F.

4.1 Physical Operators

There are two kinds of physical operators: *wrappers* and *enhancers*. A wrapper simply invokes a logical operator. Enhancers replace wrappers to take advantage of different optimization opportunities.

A wrapper invokes a logical operator together with the corresponding physical details, *e.g.,* input dataset and schema details (if available). For clarity, we discard such physical details in all the definitions below. In contrast to the logical layer, BigDansing invokes a physical operator for a set D of data units, rather than for a single unit. This enables the processing of multiple units in a single function call. For each logical operator, we define a corresponding wrapper.

(1) PScope applies a user-defined selection and projection over a set of data units D and outputs a dataset $D' \subset D$.

$$\mathsf{PScope}(D) \rightarrow \{D'\}$$

(2) PBlock takes a dataset D as input and outputs a list of key-value pairs defined by users.

$$\mathsf{PBlock}(D) \rightarrow \mathsf{map}\langle \mathsf{key}, \mathsf{list}\langle U\rangle\rangle$$

(3) PIterate takes a list of lists of Us as input and outputs their cross product or a user-defined combination.

$$\mathsf{PIterate}(\mathsf{list}\langle \mathsf{list}\langle U\rangle\rangle) \rightarrow \mathsf{list}\langle U\rangle \mid \mathsf{list}\langle \mathsf{Pair}\langle U\rangle\rangle$$

(4) PDetect receives either a list of data units Us or a list of data unit pairs U-pairs and produces a list of violations.

$$\mathsf{PDetect}(\mathsf{list}\langle U\rangle \mid \mathsf{list}\langle \mathsf{Pair}\langle U\rangle\rangle) \rightarrow \mathsf{list}\langle Violation\rangle$$

(5) PGenFix receives a list of violations as input and outputs a list of a set of possible fixes, where each set of fixes belongs to a different input violation.

$$\mathsf{PGenFix}(\mathsf{list}\langle Violation\rangle) \rightarrow \mathsf{list}\langle\{PossibleFixes\}\rangle$$

With declarative rules, the operations over a dataset are known, which enables algorithmic opportunities to improve performance. Notice that one could also discover such operations with code analysis over the UDFs [20]. However, we leave this extension to future work. BigDansing exploits such optimization opportunities via three new enhancers operators: *CoBlock*, *UCrossProduct*, and *OCJoin*. CoBlock is a physical operator that allows to group multiple datasets by a given key. UCrossProduct and OCJoin are basically two additional different implementations for the PIterate operator. We further explain these three enhancers operators in the next section.

Algorithm 1: Logical plan consolidation

```
   input  : LogicalPlan lp
   output: LogicalPlan clp
1  PlanBuilder lpb = new PlanBuilder();
2  for logical operator lop_i ∈ lp do
3      lop_j ← findMatchingLO(lop_i, lp);
4      DS_1 ← getSourceDS(lop_i);
5      DS_2 ← getSourceDS(lop_j);
6      if DS_1 == DS_2 then
7          lop_c ← getLabelsFuncs(lop_i, lop_j);
8          lop_c.setInput(DS_1, DS_2);
9          lpb.add(lop_c);
10         lp.remove(lop_i, lop_j);
11 if lpb.hasConsolidatedOps then
12     lpb.add(lp.getOperators());
13     return lpb.generateConsolidatedLP();
14 else
15     return lp;
```

4.2 From Logical to Physical Plan

As we mentioned earlier, optimizing a logical plan is performed by static analysis (plan consolidation) and by plugging enhancers (operators translation) whenever possible.

Plan Consolidation. Whenever logical operators use a different label for the same dataset, BigDansing translates them into distinct physical operators. BigDansing has to create multiple copies of the same dataset, which it might broadcast to multiple nodes. Thus, both the memory footprint at compute nodes and the network traffic are increased. To address this problem, BigDansing consolidates redundant logical operators into a single logical operator. Hence, by applying the same logical operator several times on the same set of data units using shared scans, BigDansing is able to increase data locality and reduce I/O overhead.

Algorithm 1 details the consolidation process for an input logical plan lp. For each operator lop_i, the algorithm looks for a matching operator lop_j (Lines 2-3). If lop_j has the same input dataset as lop_i, BigDansing consolidates them into lop_c (Lines 4-6). The newly consolidated operator lop_c takes the labels, functions, and datasets from lop_i and lop_j (Lines 7-8). Next, BigDansing adds lop_c into a logical plan builder lpb and removes lop_i and lop_j from lp (Lines 9-10). At last, if any operator was consolidated, it adds the non-consolidated operators to lpb and returns the consolidated logical plan (Lines 11-13). Otherwise, it returns lp (Line 15).

Let us now illustrate the consolidation process with an example. Consider a DC on the TPC-H database stating that if a customer and a supplier have the same name and phone they must be in the same city. Formally,

$$DC : \forall t_1, t_2 \in D1, \neg(t_1.c\_name = t_2.s\_name \wedge t_1.c\_phone = t_2.s\_phone \wedge t_1.c\_city \neq t_2.s\_city) \quad (1)$$

For this DC, BigDansing generates the logical plan in Figure 5(a) with operators Scope and Block applied twice over the same input dataset. It then consolidates redundant logical operators into a single one (Figure 5(b)) thereby reducing the overhead of reading an input dataset multiple times. The logical plan consolidation is only applied when it does not affect the original labeling of the operators.

Operators Translation. Once a logical plan have been consolidated, BigDansing translates the consolidated logical plan into a physical plan. It maps each logical operator to its corresponding wrapper, which in turn maps to one or more physical operators. For example, it produces the physical plan in Figure 5(c) for the consolidated logical plan in Figure 5(b). For enhancers, BigDansing exploits some particular information from the data cleansing process. Below, we detail these three enhancers operators as well as in which cases they are used by our system.

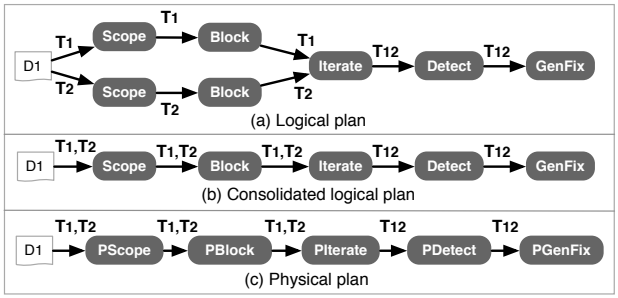


Figure 5: Plans for DC in rule 1

• *CoBlock* takes multiple input datasets $\mathbb{D}$ and applies a group-by on a given key. This would limit the comparisons required by the rule to only blocks with the same key from the different datasets. An example is shown in Figure 16 (Appendix E). Similar to the CoGroup defined in [28], in the output of CoBlock, all keys from both inputs are collected into bags. The output of this operator is a map from a key value to the list of data units sharing that key value. We formally define this operator as:

$$\mathsf{CoBlock}\ (\mathbb{D}) \rightarrow \mathsf{map}\langle \mathsf{key}, \mathsf{list}\langle \mathsf{list}\langle U \rangle \rangle \rangle$$

If two non-consolidated Block operators' outputs go to the same Iterate and then to a single PDetect, BigDansing translates them into a single CoBlock. Using CoBlock allows us to reduce the number of candidate violations for the Detect. This is because Iterate generates candidates only inside and not across CoBlocks (see Figure 6).

• *UCrossProduct* receives a single input dataset D and applies a self cross product over it. This operation is usually performed in cleansing processes that would output the same violations irrespective of the order of the input of Detect. UCrossProduct avoids redundant comparisons, reducing the number of comparisons from n^2 to $\frac{n \times (n-1)}{2}$, with n being the number of units Us in the input. For example, the output of the logical operator Iterate in Figure 2 is the result of UCrossProduct within each block; there are four pairs instead of thirteen, since the operator avoids three comparisons for the elements in block B1 and six for the ones in B3 of Figure 2. Formally:

$$\mathsf{UCrossProduct}(D) \rightarrow \mathsf{list}\langle \mathsf{Pair}\langle U \rangle \rangle$$

If, for a single dataset, the declarative rules contain only symmetric comparisons, *e.g.,* $=$ and $\neq$, then the order in which the tuples are passed to PDetect (or to the next logical operator if any) does not matter. In this case, BigDansing uses UCrossProduct to avoid materializing many unnecessary pairs of data units, such as for the Iterate operator in Figure 2. It also uses UCrossProduct when: (i) users do not provide a matching Block operator for the Iterate operator; or (ii) users do not provide any Iterate or Block operator.

• *OCJoins* performs a self join on one or more ordering comparisons (*i.e.,* $<, >, \geq, \leq$). This is a very common operation

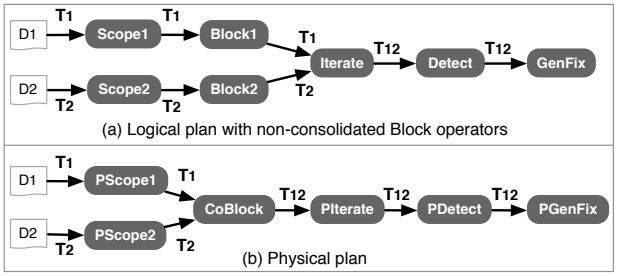


Figure 6: Example plans with CoBlock

in rules such as DCs. Thus, BIGDANSING provides OCJoin, which is an efficient operator to deal with join ordering comparisons. It takes input dataset D and applies a number of join conditions, returning a list of joining U-pairs. We formally define this operator as follows:

$$\mathsf{OCJoin}(D) \rightarrow \mathsf{list}\langle \mathsf{Pair}\langle U \rangle \rangle$$

Every time BIGDANSING recognizes joins conditions defined with ordering comparisons in PDetect, *e.g.,* ϕ_D, it translates Iterate into a OCJoin implementation. Then, it passes OCJoin output to a PDetect operator (or to the next logical operator if any). We discuss this operator in detail in Section 4.3.

We report details for the transformation of the physical operators to two distributed platforms in Appendix G.

4.3 Fast Joins with Ordering Comparisons

Existing systems handle joins over ordering comparisons using a cross product and a post-selection predicate, leading to poor performance. BIGDANSING provides an efficient ad-hoc join operator, referred to as OCJoin, to handle these cases. The main goal of OCJoin is to increase the ability to process joins over ordering comparisons in parallel and to reduce its complexity by reducing the algorithm's search space. In a nutshell, OCJoin first range partitions a set of data units and sorts each of the resulting partitions in order to validate the inequality join conditions in a distributed fashion. OCJoin works in four main phases: *partitioning*, *sorting*, *pruning*, and *joining* (see Algorithm 2).

Partitioning. OCJoin first selects the attribute, $PartAtt$, on which to partition the input D (line 1). We assume that all join conditions have the same output cardinality. This can be improved using cardinality estimation techniques [9,27], but it is beyond the scope of the paper. OCJoin chooses the first attribute involved in the first condition. For instance, consider again ϕ_D (Example 1), OCJoin sets $PartAtt$ to rate attribute. Then, OCJoin partitions the input dataset D into $nbParts$ range partitions based on $PartAtt$ (line 2). As part of this partitioning, OCJoin distributes the resulting partitions across all available computing nodes. Notice that OCJoin runs the range partitioning in parallel. Next, OCJoin forks a parallel process for each range partition k_i to run the three remaining phases (lines 3-14).

Sorting. For each partition, OCJoin creates as many sorting lists ($Sorts$) as inequality conditions are in a rule (lines 4-5). For example, OCJoin creates two sorted lists for ϕ_D: one sorted on rate and the other sorted on salary. Each list contains the attribute values on which the sort order is and the tuple identifiers. Note that OCJoin only performs a local sorting in this phase and hence it does not require any data transfer across nodes. Since multiple copies of a partition may exist in multiple computing nodes, we apply *sorting* before *pruning* and *joining* phases to ensure that each partition is sorted at most once.

Algorithm 2: OCJoin

```
   input : Dataset D, Condition conds[], Integer nbParts
   output: List Tuples⟨Tuple⟩
   // PARTITIONING PHASE
1  PartAtt ← getPrimaryAtt(conds[].getAttribute());
2  K ← RangePartition(D, PartAtt, nbParts);
3  for each k_i ∈ K do
4      for each c_j ∈ conds[] do // SORTING
5          Sorts[j] ← sort(k_i, c_j.getAttribute());
6      for each k_l ∈ {k_{i+1}...k_{|K|}} do
7          if overlap(k_i, k_l, PartAtt) then // PRUNING
8              tuples = ∅;
9              for each c_j ∈ conds[] do // JOINING
10                 tuples ← join(k_i, k_l, Sorts[j], tuples);
11                 if tuples == ∅ then
12                     break;
13             if tuples != ∅ then
14                 Tuples.add(tuples);
```

Pruning. Once all partitions $k_i \in K$ are internally sorted, OCJoin can start joining each of these partitions based on the inequality join conditions. However, this would require transferring large amounts of data from one node to another. To circumvent such an overhead, OCJoin inspects the min and max values of each partition to avoid joining partitions that do not overlap in their min and max range (the *pruning* phase, line 7). Non-overlapping partitions do not produce any join result. If the selectivity values for the different inequality conditions are known, OCJoin can order the different joins accordingly.

Joining. OCJoin finally proceeds to join the overlapping partitions and outputs the join results (lines 9-14). For this, it applies a distributed sort merge join over the sorted lists, where some partitions are broadcast to other machines while keeping the rest locally. Through pruning, OCJoin tells the underlying distributed processing platform which partitions to join. It is up to that platform to select the best approach to minimize the number of broadcast partitions.

5. DISTRIBUTED REPAIR ALGORITHMS

Most of the existing repair techniques [5–7, 11, 17, 23, 34] are centralized. We present two approaches to implement a repair algorithm in BIGDANSING. First, we show how our system can run a centralized data repair algorithm in parallel, without changing the algorithm. In other words, BIGDANSING treats that algorithm as a black box (Section 5.1). Second, we design a distributed version of the widely used *equivalence class* algorithm [5, 7, 11, 17] (Section 5.2).

5.1 Scaling Data Repair as a Black Box

Overall, we divide a repair task into independent smaller repair tasks. For this, we represent the violation graph as a hypergraph containing the violations and their possible fixes. The nodes represent the elements and each hyperedge covers a set of elements that together violate a rule, along with possible repairs. We then divide the hypergraph into smaller independent subgraphs, *i.e., connected components*,

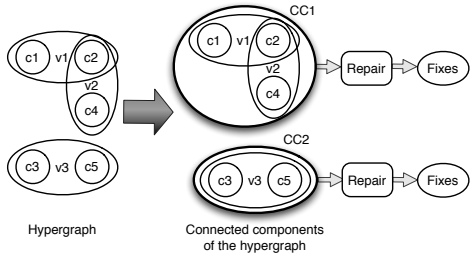


Figure 7: Data repair as a black box.

and we pass each connected component to an *independent data repair instance*.

Connected components. Given an input set of violations, form at least one rule, BigDansing first creates their hypergraph representation of such possible fixes in a distributed manner. It then uses GraphX [37] to find all connected components in the hypergraph. GraphX, in turn, uses the Bulk Synchronous Parallel (BSP) graph processing model [25] to process the hypergraph in parallel. As a result, BigDansing gets a connected component ID for each hyperedge. It then groups hyperedges by the connected component ID. Figure 7 shows an example of connected components in a hypergraph containing three violations $v1$, $v2$, and $v3$. Note that violations $v1$ and $v2$ can be caused by different rules. Violations $v1$ and $v2$ are grouped in a single connected component $CC1$, because they share element $c2$. In contrast, $v3$ is assigned to a different connected component $CC2$, because it does not share any element with $v1$ or $v2$.

Independent data repair instance. Once all connected components are computed, BigDansing assigns each of them to an independent data repair instance and runs such repair instances in a distributed manner (right-side of Figure 7). When all data repair instances generate the required fixes, BigDansing updates the input dataset and pass it again to the RuleEngine to detect potential violations introduced by the RepairAlgorithm. The number of iterations required to fix all violations depends on the input rules, the dataset, and the repair algorithm.

Dealing with big connected components. If a connected component does not fit in memory, BigDansing uses a k-way multilevel hypergraph partitioning algorithm [22] to divide it into k equal parts and run them on distinct machines. Unfortunately, naively performing this process can lead to inconsistencies and contradictory choices in the repair. Moreover, it can fix the same violation independently in two machines, thus introducing unnecessary changes to the repair. We illustrate this problem with an example.

Example 2: Consider a relational schema D with 3 attributes A, B, C and 2 FDs $A \rightarrow B$ and $C \rightarrow B$. Given the instance $[t_1]$(a1, b1, c1), $[t_2]$(a1, b2, c1), all data values are in violation: $t_1.A$, $t_1.B$, $t_2.A$, $t_2.B$ for the first FD and $t_1.B$, $t_1.C$, $t_2.B$, $t_2.C$ for the second one. Assuming the compute nodes have enough memory only for five values, we need to solve the violations by executing two instances of the algorithm on two different nodes. Regardless of the selected tuple, suppose the first compute node repairs a value on attribute A, and the second one a value on attribute C. When we put the two repairs together and check their consistency, the updated instance is a valid solution, but the repair is not minimal because a single update on attribute B would have solved both violations. However, if the first node fixes $t_1.B$ by assigning value "b2" and the second one fixes $t_2.B$ with "b1", not only there are two changes, but the final instance is also inconsistent. □

We tackle the above problem by assigning the role of *master* to one machine and the role of *slave* to the rest. Every machine applies a repair in isolation, but we introduce an extra test in the union of the results. For the violations that are solved by the master, we mark its changes as immutable, which prevents us to change an element more than once. If a change proposed by a slave contradicts a possible repair that involve a master's change, the slave repair is undone and a new iteration is triggered. As a result, the algorithm always reaches a fix point to produce a clean dataset, because an updated value cannot change in the following iterations.

BigDansing currently provides two repair algorithms using this approach: the equivalence class algorithm and a general hypergraph-based algorithm [6, 23]. Users can also implement their own repair algorithm if it is compatible with BigDansing's repair interface.

5.2 Scalable Equivalence Class Algorithm

The idea of the equivalence class based algorithm [5] is to first group all elements that should be equivalent together, and to then decide how to assign values to each group. An *equivalence class* consists of pairs of the form (t, A), where t is a data unit and A is an element. In a dataset D, each unit t and each element A in t have an associated equivalence class, denoted by $\mathsf{eq}(t, A)$. In a repair, a unique *target value* is assigned to each equivalence class E, denoted by $\mathsf{targ}(E)$. That is, for all $(t, A) \in E$, $t[A]$ has the same value $\mathsf{targ}(E)$. The algorithm selects the target value for each equivalence class to obtain a repair with the minimum overall cost.

We extend the equivalence class algorithm to a distributed setting by modeling it as a distributed word counting algorithm based on map and reduce functions. However, in contrast to a standard word count algorithm, we use two map-reduce sequences. The first map function maps the violations' possible fixes for each connected component into key-value pairs of the form ⟨⟨ccID,value⟩,count⟩. The key ⟨ccID,value⟩ is a composite key that contains the connected component ID and the element value for each possible fix. The value *count* represents the frequency of the element value, which we initialize to 1. The first *reduce* function counts the occurrences of the key-value pairs that share the same connected component ID and element value. It outputs key-value pairs of the form ⟨⟨ccID,value⟩,count⟩. Note that if an element exists in multiple fixes, we only count its value once. After this first map-reduce sequence, another map function takes the output of the reduce function to create new key-value pairs of the form ⟨ccID,⟨value,count⟩⟩. The key is the connected component ID and the value is the frequency of each element value. The last reduce selects the element value with the highest frequency to be assigned to all the elements in the connected component $ccID$.

6. EXPERIMENTAL STUDY

We evaluate BigDansing using both real and synthetic datasets with various rules (Section 6.1). We consider a variety of scenarios to evaluate the system and answer the following questions: (i) how well does it perform compared with baseline systems in a single node setting (Section 6.2)? (ii) how well does it scale to different dataset sizes compared to the state-of-the-art distributed systems (Section 6.3)? (iii) how well does it scale in terms of the number of nodes

Dataset	Rows	Dataset	Rows
$TaxA_1$–$TaxA_5$	100K – 40M	customer2	32M
$TaxB_1$–$TaxB_3$	100K – 3M	NCVoter	9M
$TPCH_1$–$TPCH_{10}$	100K – 1907M	HAI	166k
customer1	19M		

Table 2: Statistics of the datasets

Identifier	Rule
φ_1 (FD):	Zipcode → City
φ_2 (DC):	$\forall t_1, t_2 \in TaxB, \neg(t_1.\text{Salary} > t2.\text{Salary} \wedge t_1.\text{Rate} < t_2.\text{Rate})$
φ_3 (FD):	o_custkey → c_address
φ_4 (UDF):	Two rows in Customer are duplicates
φ_5 (UDF):	Two rows in NCVoter are duplicates
φ_6 (FD):	Zipcode → State
φ_7 (FD):	PhoneNumber → Zipcode
φ_8 (FD):	ProviderID → City, PhoneNumber

Table 3: Integrity constraints used for testing

(Section 6.4)? (iv) how well does its abstraction support a variety of data cleansing tasks, e.g., for deduplication (Section 6.5)? and (v) how do its different techniques improve performance and what is its repair accuracy (Section 6.6)?

6.1 Setup

Table 2 summarizes the datasets and Table 3 shows the rules we use for our experiments.

(1) TaxA. Represents personal tax information in the US [11]. Each row contains a person's name, contact information, and tax information. For this dataset, we use the FD rule φ_1 in Table 3. We introduced errors by adding random text to attributes City and State at a 10% rate.

(2) TaxB. We generate TaxB by adding 10% numerical random errors on the Rate attribute of TaxA. Our goal is to validate the efficiency with rules that have inequality conditions only, such as DC φ_2 in Table 3.

(3) TPCH. From the TPC-H benchmark data [2], we joined the lineitem and customer tables and applied 10% random errors on the address. We use this dataset to test FD rule φ_3.

(4) Customer. In our deduplication experiment, we use TPC-H customer with 4.5 million rows to generate tables customer1 with 3x exact duplicates and customer2 with 5x exact duplicates. Then, we randomly select 20% of the total number of tuples, in both relations, and duplicate them with random edits on attributes name and phone.

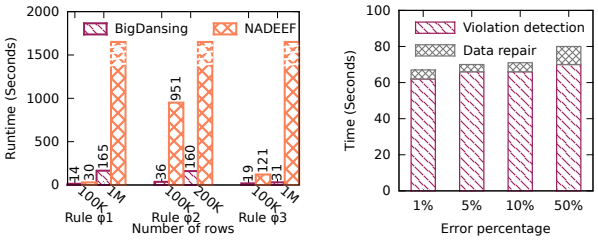
(5) NCVoter. This is a real dataset that contains North Carolina voter information. We added 20% random duplicate rows with random edits in name and phone attributes.

(6) Healthcare Associated Infections (HAI) (http://www.hospitalcompare.hhs.gov). This real dataset contains hospital information and statistics measurements for infections developed during treatment. We added 10% random errors on the attributes covered by the FDs and tested four combinations of rules (φ_6 – φ_8 from Table 3). Each rule combination has its own dirty dataset.

To our knowledge, there exists only one full-fledged data cleansing system that can support all the rules in Table 3:

(1) NADEEF [7]: An open-source single-node platform supporting both declarative and user defined quality rules.

(2) PostgreSQL v9.3: Since declarative quality rules can be represented as SQL queries, we also compare to PostgreSQL for violation detection. To maximize benefits from large main memory, we configured it using pgtune [30].



(a) Rules φ_1, φ_2, and φ_3. (b) Rule φ_1.

Figure 8: Data cleansing times.

The two declarative constraints, DC φ_2 and FD φ_3 in Table 3, are translated to SQL as shown below.

φ_2:	SELECT a.Salary, b.Salary, a.Rate, b.Rate FROM TaxB a JOIN TaxB b WHERE a.Salary > b.Salary AND a.Rate < b.Rate;
φ_3:	SELECT a.o_custkey, a.c_address, b.c_address FROM TPCH a JOIN TPCH b ON a.custkey = b.custkey WHERE a.c_address ≠ b.c_address;

We also consider two parallel data processing frameworks:

(3) Shark 0.8.0 [38]: This is a scalable data processing engine for fast data analysis. We selected Shark due to its scalability advantage over existing distributed SQL engines.

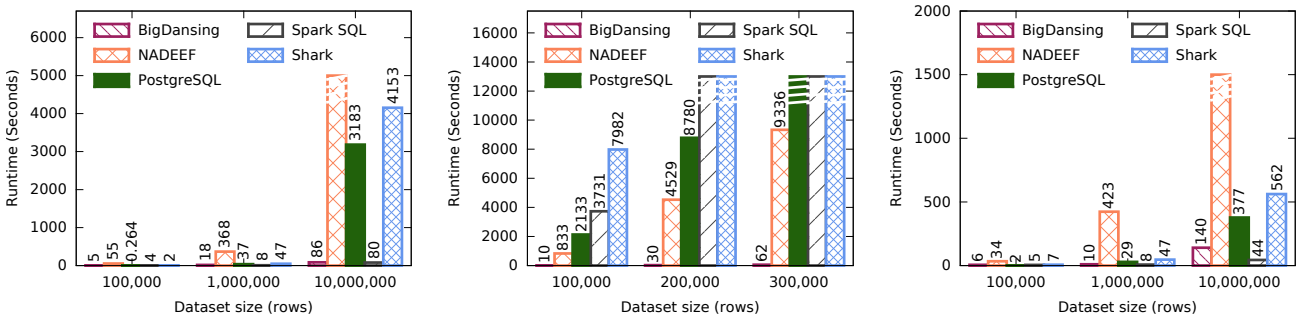
(4) Spark SQL v1.0.2:It is an experimental extension of Spark v1.0 that allows users run relational queries natively on Spark using SQL or HiveQL. We selected this system as, like BigDansing, it runs natively on top of Spark.

We ran our experiments in two different settings: (i) a single-node setting using a Dell Precision T7500 with two 64-bit quad-core Intel Xeon X5550 (8 physical cores and 16 CPU threads) and 58GB RAM and (ii) a multi-node setting using a compute cluster of 17 Shuttle SH55J2 machines (1 master with 16 workers) equipped with Intel i5 processors with 16GB RAM constructed as a star network.

6.2 Single-Node Experiments

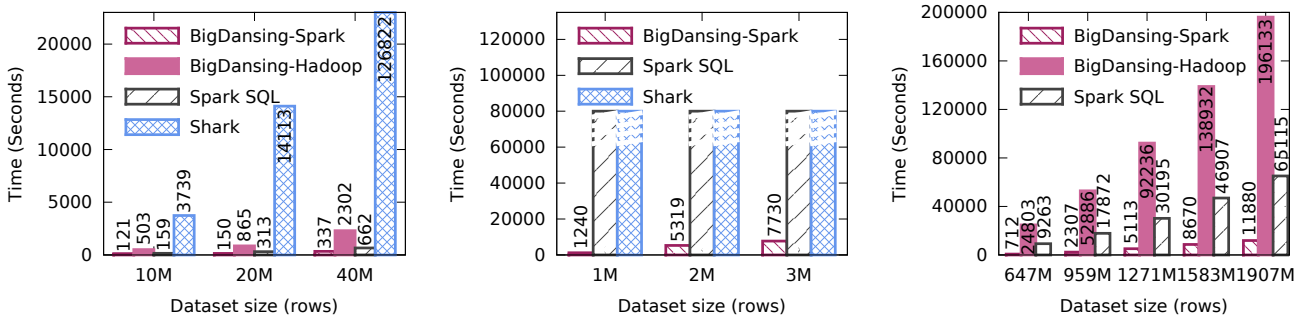
For these experiments, we start comparing BigDansing with NADEEF in the execution times of the whole cleansing process (*i.e.,* detection and repair). Figure 8(a) shows the performance of both systems using the TaxA and TPCH datasets with 100K and 1M (200K for φ_2) rows. We observe that BigDansing is more than three orders of magnitude faster than NADEEF in rules φ_1 (1M) and φ_2 (200K), and φ_3 (1M). In fact, NADEEF is only "competitive" in rule φ_1 (100K), where BigDansing is only twice faster. The high superiority of BigDansing comes from two main reasons: (i) In contrast to NADEEF, it provides a finer granular abstraction allowing users to specify rules more efficiently; and (ii) It performs rules with inequality conditions in an efficient way (using OCJoin). In addition, NADEEF issues thousands of SQL queries to the underlying DBMS. for detecting violations. We do not report results for larger datasets because NADEEF was not able to run the *repair* process for more than 1M rows (300K for φ_2).

We also observed that violation detection was dominating the entire data cleansing process. Thus, we ran an experiment for rule φ_1 in TaxA with 1M rows by varying the error rate. Violation detection takes more than 90% of the time, regardless of the error rate (Figure 8(b)). In particular, we observe that even for a very high error rate (50%), the violation detection phase still dominates the cleansing process. Notice that for more complex rules, such as rule φ_2, the dominance of the violation detection is even ampli-



(a) TaxA data with φ_1 (b) TaxB data with φ_2 (c) TPCH data with φ_3

Figure 9: Single-node experiments with φ_1, φ_2, and φ_1



(a) TaxA data with φ_1 (b) TaxB data with φ_2 (c) Large TPCH on φ_3

Figure 10: Multi-nodes experiments with φ_1, φ_2 and φ_3

fied. Therefore, to be able to extensively evaluate BigDansing, we continue our experiments focusing on the violation detection phase only, except if stated otherwise.

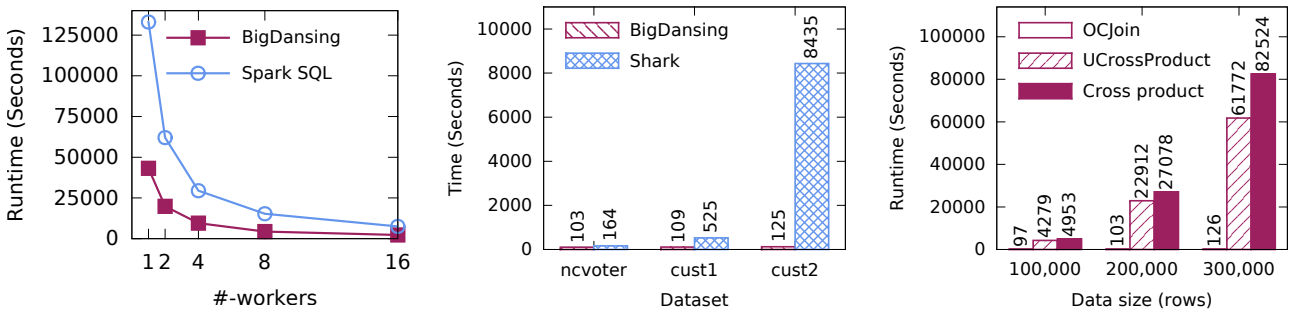
Figures 9(a), Figure 9(b), and 9(c) show the violation detection performance of all systems for TaxA, TaxB, and TPCH datasets. TaxA and TPCH datasets have 100K, 1M, and 10M rows and TaxB has 100K, 200K, and 300K rows. With equality-based rules φ_1 and φ_3, we observe that PostgreSQL is always faster than all other systems on the small datasets (100K rows). However, once we increase the size by one order of magnitude (1M rows), we notice the advantage of BigDansing: it is at least twice faster than PostgreSQL and more than one order of magnitude faster than NADEEF. For the largest dataset (10M rows), BigDansing is almost two orders of magnitude faster than PostgreSQL for the FD in TaxA and one order of magnitude faster than PostgreSQL for the DC in TaxA. For the FD in TPCH, BigDansing is twice faster than PostgreSQL. It is more than three orders of magnitude faster than NADEEF for all rules. Overall, we observe that BigDansing performs similarly to Spark SQL. The small difference is that Spark SQL uses multithreading better than BigDansing. We can explain the superiority of BigDansing compared to the other systems along two reasons: (i) BigDansing reads the input dataset only once, while PostgreSQL and Shark read it twice because of the self joins; and (ii) BigDansing does not generate duplicate violations, while SQL engines do when comparing tuples using self-joins, such as for TaxA and TPCH's FD. Concerning the inequality-based rule φ_2, we limited the runtime to four hours for all systems. BigDansing is one order of magnitude faster than all other systems for 100K rows. For 200K and 300K rows, it is at least two orders of magnitude faster than all baseline systems. Such performance superiority is achieved by leveraging the inequality join optimization that is not supported by the baseline systems.

6.3 Multi-Node Experiments

We now compare BigDansing with Spark SQL and Shark in the multi-node setting. We also implemented a lighter version of BigDansing on top of Hadoop MapReduce to show BigDansing independence *w.r.t.* the underlying framework. We set the size of TaxA to 10M, 20M, and 40M rows. Moreover, we tested the inequality DC φ_2 on TaxB dataset with sizes of 1M, 2M, and 3M rows. We limited the runtime to 40 hours for all systems.

BigDansing-Spark is slightly faster than Spark SQL for the equality rules in Figure 10(a). Even though BigDansing-Spark and Shark are both implemented on top of Spark, BigDansing-Spark is up to three orders of magnitude faster than Shark. Even BigDansing-Hadoop is doing better than Shark (Figure 10(a)). This is because Shark does not process joins efficiently. The performance of BigDansing over baseline systems is magnified when dealing with inequalities. We observe that BigDansing-Spark is at least two orders of magnitude faster than both Spark SQL and Shark (Figure 10(b)). We had to stop Spark SQL and Shark executions after 40 hours of runtime; both Spark SQL and Shark are unable to process the inequality DC efficiently.

We also included a testing for large TPCH datasets of sizes 150GB, 200GB, 250GB, and 300GB (959M, 1271M, 1583M, and 1907M rows resp.) producing between 6.9B and 13B violations. We excluded Shark as it could not run on these larger datasets. BigDansing-Spark is 16 to 22 times faster than BigDansing-Hadoop and 6 to 8 times faster than Spark SQL (Figure 10(c)). BigDansing-Spark significantly outperforms Spark SQL since it has a lower I/O complexity and an optimized data cleansing physical execution plan. Moreover, the performance difference between BigDansing-Spark and BigDansing-Hadoop stems from Spark being generally faster than Hadoop; Spark is an in-memory data processing system while Hadoop is disk-based.



(a) 500M rows TPCH on φ_3 (b) Deduplication in BigDansing (c) OCJoin and UCrossProdut

Figure 11: Results for the experiments on (a) scale-out, (b) deduplication, and (c) physical optimizations.

6.4 Scaling BigDansing Out

We compared the speedup of BigDansing-Spark to Spark SQL when increasing the number of workers on a dataset size of 500M rows. We observe that BigDansing-Spark is at least 3 times faster than Spark SQL starting from one single worker and up to 16 (Figure 11(a)). We also notice that BigDansing-Spark is about 1.5 times faster than Spark SQL while using only half the number of workers used by Spark SQL. Although both BigDansing-Spark and Spark SQL generally have a good scalability, BigDansing-Spark performs better than Spark SQL on large input datasets since it does not copy the input data twice.

6.5 Deduplication in BigDansing

We show that one can run a deduplication task with BigDansing. We use *cust1*, *cust2*, and *NCVoters* datasets on our compute cluster. We implemented a Java version of Levenshtein distance and use it as a UDF in both BigDansing and Shark. Note that we do not consider Spark SQL in this experiment since UDFs cannot be implemented directly within Spark SQL. That is, to implement a UDF in Spark SQL the user has to either use a Hive interface or apply a post processing step on the query result. Figure 11(b) shows the results. We observe that BigDansing outperforms Shark for both small datasets as well as large datasets. In particular, we see that for cust2 BigDansing outperforms Shark up to an improvement factor of 67. These results not only show the generality of BigDansing supporting a deduplication task, but also the high efficiency of our system.

6.6 BigDansing In-Depth

Physical optimizations. We first focus on showing the benefits in performance of the UCrossProduct and OCJoin operators. We use the second inequality DC in Table 3 with TaxB dataset on our multi-node cluster. Figure 11(c) reports the results of this experiment. We notice that the UCrossProduct operator has a slight performance advantage compared to the CrossProduct operator. This performance difference increases with the dataset size. However, by using the OCJoin operator, BigDansing becomes more than two orders of magnitudes faster compared to both cross product operators (up to an improvement factor of 655).

Abstraction advantage. We now study the benefits of BigDansing's abstraction. We consider the deduplication scenario in previous section with the smallest TaxA dataset on our single-node machine. We compare the performance difference between BigDansing using its full API and BigDansing using only the Detect operator. We see in Figure 12(a) that running a UDF using the full BigDansing API makes BigDansing three orders of magnitudes faster compared to using *Detect* only. This clearly shows the benefits of the five logical operators, even for single-node settings.

Scalable data repair. We study the runtime efficiency of the repair algorithms used by BigDansing. We ran an experiment for rule φ_1 in TaxA with 1M rows by varying the error rate and considering two versions of BigDansing: the one with the parallel data repair and a baseline with a centralized data repair, such as in NADEEF. Figure 12(b) shows the results. The parallel version outperforms the centralized one, except when the error rate is very small (1%). For higher rates, BigDansing is clearly faster, since the number of connected components to deal with in the repair process increases with the number of violations and thus the parallelization provides a stronger boost. Naturally, our system scales much better with the number of violations.

Repair accuracy. We evaluate the accuracy of BigDansing using the traditional *precision* and *recall* measure: *precision* is the ratio of correctly updated attributes (exact matches) to the total number of updates; *recall* is the ratio of correctly updated attributes to the total number of errors.

We test BigDansing with the equivalence class algorithm using HAI on the following rule combinations: (a) FD φ_6; (b) FD φ_6 and FD φ_7; (c) FD φ_6, FD φ_7, and FD φ_8. Notice that BigDansing runs (a)-combination alone while it runs (b)-combination and (c)-combination concurrently.

Table 4 shows the results for the equivalence class algorithm in BigDansing and NADEEF. We observe that BigDansing achieves similar accuracy and recall as the one obtained in a centralized system, *i.e.,* NADEEF. In particular, BigDansing requires the same number of iterations as NADEEF even to repair all data violations when running multiple rules concurrently. Notice that both BigDansing and NADEEF might require more than a single iterations when running multiple rules because repairing some violations might cause new violations. We also tested the equivalence class in BigDansing using both the black box and scalable repair implementation, and both implementations achieved similar results.

We also test BigDansing with the hypergraph algorithm using DC rule ϕ_D on a TaxB dataset. As the search space of the possible solutions in ϕ_D is huge, the hypergraph algorithm uses quadratic programming to approximates the repairs in ϕ_D [6]. Thus, we use the euclidean distance to measure the repairs accuracy on the repaired data attributes compared to the attributes of the ground truth.

Rule(s)	NADEEF		BigDansing		Iterations
	precision	recall	precision	recall	
φ_6	0.713	0.776	0.714	0.777	1
φ_6&φ_7	0.861	0.875	0.861	0.875	2
φ_6–φ_8	0.923	0.928	0.924	0.929	3
	|R,G|/e	**|R,G|**	**|R,G|/e**	**|R,G|**	**Iter.**
ϕ_D	17.1	8183	17.1	8221	5

Table 4: Repair quality using HAI and TaxB datasets.

Table 4 shows the results for the hypergraph algorithm in BigDansing and NADEEF. Overall, we observe that BigDansing achieves the same average distance ($|R,G|/e$) and similar total distance ($|R,G|$) between the repaired data R and the ground truth G. Again, BigDansing requires the same number of iterations as NADEEF to completely repair the input dataset. These results confirm that BigDansing achieves the same data repair quality as in a single node setting. This by providing better data cleansing runtimes and scalability than baselines systems.

7. RELATED WORK

Data cleansing, also called data cleaning, has been a topic of research in both academia and industry for decades, *e.g.,* [6, 7, 11–15, 17–19, 23, 29, 34, 36]. Given some "dirty" dataset, the goal is to find the most likely errors and a possible way to repair these errors to obtain a "clean" dataset. In this paper, we are interested in settings where the detection of likely errors and the generation of possible fixes is expressed through UDFs. As mentioned earlier, traditional constraints, *e.g.,* FDs, CFDs, and DCs, are easily expressed in BigDansing and hence any errors detected by these constraints will be detected in our framework. However, our goal is not to propose a new data cleansing algorithm but rather to provide a framework where data cleansing, including detection and repair, can be performed at scale using a flexible UDF-based abstraction.

Work in industry, *e.g.,* IBM QualityStage, SAP BusinessObjects, Oracle Enterprise Data Quality, and Google Refine has mainly focused on the use of low-level ETL rules [3]. These systems do not support UDF-based quality rules in a scalable fashion as in BigDansing.

Examples of recent work in data repair include cleaning algorithms for DCs [6, 17] and other fragments, such as FDs and CFDs [4, 5, 7, 11, 17]. These proposals focus on specific logical constraints in a centralized setting, without much regards to scalability and flexibility as in BigDansing. As shown in Section 5, we adapted two of these repair algorithms to our distributed platform.

Another class of repair algorithms use machine learning tools to clean the data. Examples include SCARE [39] and ERACER [26]. There are also several efforts to include users (experts or crowd) in the cleaning process [33,40]. Both lines of research are orthogonal to BigDansing.

Closer to our work is NADEEF [7], which is a generalized data cleansing system that detects violations of various data quality rules in a unified programming interface. In contrast to NADEEF, BigDansing: (i) provides a richer programming interface to enable efficient and scalable violation detection, *i.e., block*(), *scope*(), and *iterate*(), (ii) enables several optimizations through its plans, (iii) introduces the first distributed approaches to data repairing.

SampleClean [35] aims at improving the accuracy of aggregate queries by performing cleansing over small samples of the source data. SampleClean focuses on obtaining unbiased query results with confidence intervals, while BigDansing focuses on providing a scalable framework for data cleansing. In fact, one cannot use SampleClean in traditional query processing where the entire input dataset is required.

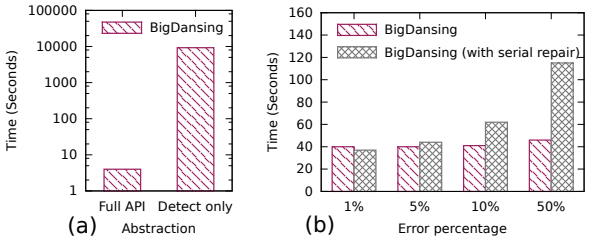


Figure 12: (a) Abstraction and (b) scalable repair.

As shown in the experiment section, scalable data processing platform, such as MapReduce [8] or Spark [41], can implement the violation detection process. However, coding the violation detection process on top of these platforms is a tedious task and requires technical expertise. We also showed that one could use a declarative system (*e.g.,* Hive [32], Pig [28], or Shark [1]) on top of one of these platforms and re-implement the data quality rules using its query language. However, many common rules, such as rule ϕ_U, go beyond declarative formalisms. Finally, these frameworks do not natively support inequality joins.

Implementing efficient theta-joins in general has been largely studied in the database community [9, 27]. Studies vary from low-level techniques, such as minimizing disk accesses for band-joins by choosing partitioning elements using sampling [9], to how to map arbitrary join conditions to Map and Reduce functions [27]. These proposals are orthogonal to our OCJoin algorithm. In fact, in our system, they rely at the executor level: if they are available in the underlying data processing platform, they can be exploited when translating the OCJoin operator at the physical level.

Dedoop [24] detects duplicates in relational data using Hadoop. It exploits data blocking and parallel execution to improve performance. Unlike BigDansing, this service-based system maintains its own data partitioning and distribution across workers. Dedoop does not provide support for scalable validation of UDFs, nor repair algorithms.

8. CONCLUSION

BigDansing, a system for fast and scalable big data cleansing, enables ease-of-use through a user-friendly programming interface. Users use logical operators to define rules which are transformed into a physical execution plan while performing several optimizations. Experiments demonstrated the superiority of BigDansing over baseline systems for different rules on both real and synthetic data with up to two order of magnitudes improvement in execution time without sacrificing the repair quality. Moreover, BigDansing is scalable, *i.e.,* it can detect violation on 300GB data (1907M rows) and produce 1.2TB (13 billion) violations in a few hours.

There are several future directions. One relates to abstracting the repairing process through logical operators, similar to violation detection. This is challenging because most of the existing repair algorithms use different heuristics to find an "optimal" repair. Another direction is to exploit opportunities for multiple data quality rule optimization.

9. REFERENCES

[1] Shark (Hive on Spark). `https://github.com/amplab/shark`.

[2] TPC-H benchmark version 2.14.4. `http://www.tpc.org/tpch/`.

[3] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[4] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3:197–207, 2010.

[5] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[6] X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, 2013.

[7] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, 2013.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, 1991.

[10] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool, 2012.

[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6:1–6:48, 2008.

[12] W. Fan, F. Geerts, N. Tang, and W. Yu. Conflict resolution with data currency and consistency. *J. Data and Information Quality*, 5(1-2):6, 2014.

[13] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[14] W. Fan, J. Li, N. Tang, and W. Yu. Incremental Detection of Inconsistencies in Distributed Data. In *ICDE*, 2012.

[15] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353), 1976.

[16] T. Friedman. Magic quadrant for data quality tools. `http://www.gartner.com/`, 2013.

[17] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 6(9):625–636, 2013.

[18] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, 2014.

[19] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[20] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4(6):385–396, 2011.

[21] A. Jindal, J.-A. Quiané-Ruiz, and S. Madden. Cartilage: Adding Flexibility to the Hadoop Skeleton. In *SIGMOD*, 2013.

[22] G. Karypis and V. Kumar. Multilevel K-way Hypergraph Partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC. ACM, 1999.

[23] S. Kolahi and L. V. S. Lakshmanan. On Approximating Optimum Repairs for Functional Dependency Violations. In *ICDT*, 2009.

[24] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 2012.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, 2010.

[26] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[27] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, 2008.

[29] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[30] G. Smith. *PostgreSQL 9.0 High Performance: Accelerate your PostgreSQL System and Avoid the Common Pitfalls that Can Slow it Down*. Packt Publishing, 2010.

[31] N. Swartz. Gartner warns firms of 'dirty data'. *Information Management Journal*, 41(3), 2007.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[33] Y. Tong, C. C. Cao, C. J. Zhang, Y. Li, and L. Chen. CrowdCleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *ICDE*, 2014.

[34] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[35] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A Sample-and-Clean Framework for Fast and Accurate Query Processing on Dirty Data. In *SIGMOD*, 2014.

[36] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[37] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES. ACM, 2013.

[38] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[39] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[40] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

APPENDIX

A. JOB EXAMPLE

```
1  BigDansing job = new BigDansing("Example Job" );
2  String  schema = "name,zipcode,city, state , salary , rate";
3  job.addInputPath(schema,"D1", "S",  "T");
4  job.addInputPath(schema,"D2", "W");
5  job.addScope(Scope, "S");
6  job.addBlock(Block1, "S");
7  job.addBlock(Block2, "T");
8  job. addIterate ("M",  Iterate1 ,  "S",  "T");
9  job. addIterate ("V",  Iterate2 ,  "W", "M");
10 job.addDetect(Detect, "V");
11 job.addGenFix(GenFix,"V");
12 job.run();
}
```

Listing 3: Example of a user's BigDansing job.

Let us explain with a BigDansing job example how users can specify to BigDansing in which sequence to run their logical operators. Users can fully control the execution flow of their logical operators via *data labels*, which in other words represent the data flow of a specific input dataset. For example, users would write the BigDansing job in Listing 3 to generate the logical plan in Section 3.2. First of all, users create a job instance to specify BigDansing their requirements (Line 1). Users specify the input datasets they want

to process by optionally providing their schema (Line 2) and their path (D1 and D2 in Lines 3 & 4). Additionally, users label input datasets to defines the number data flows they desire to have (S and T for D1 and W for D2). Notice that, in this example, the job creates a copy of D1. Then, users specify the sequence of logical operations they want to perform to each data flow (Lines 5-11). BIGDANSING respect the order in which users specify their logical operators, e.g., BIGDANSING will first perform Scope and then Block1. Users can also specify arbitrary number of inputs and outputs. For instance, in Iterate1 get data flows S and T as input and outputs a signal data flow M. As last step, users run their job as in Line 12.

B. LOGICAL OPERATORS LISTINGS

We show in Listing 4 the code for Scope, Listing 5 the code for Block and in Listing 6 the code for Iterate in rule ϕ_F.

```
public Tuple scope(Tuple in) {
1     Tuple t = new Tuple(in.getID());
2     t.addCell(in.getCellValue(1));// zipcode
3     t.addCell(in.getCellValue(2));// city
4     return t;   }
```

Listing 4: Code example for the Scope operator.

```
public String block(Tuple t) {
1     return t.getCellValue(0);// zipcode   }
```

Listing 5: Code example for the Block operator.

```
public Iterable <TuplePair> iterator( ListTupleList <String> in) {
1     ArrayList<TuplePair> tp = new ArrayList<TuplePair>();
2     List<Tuple> inList = in.getIterator().next().getValue();
3     for (int i = 0; i < inList.size(); i++) {
4        for (int j = i + 1; j < inList.size(); j++) {
5           tp.add(new TuplePair(inList.get(i), inList.get(j)));
6        }
7     }
8     return tp;   }
```

Listing 6: Code example for the Iterate operator.

C. RDF DATA REPAIR

Let us explain how BIGDANSING works for RDF data cleansing with an example. Consider an RDF dataset containing students, professors, and universities, where each student is enrolled in one university and has one professor as advisor. The top-left side of Figure 13 shows this RDF dataset as set of triples (RDF input) and Figure 14 shows the graph representation of such an RDF dataset. Let's assume that there cannot exist two graduate students in two different universities and have the same professor as advisor. According to this rule, there are two violations in this RDF dataset: (Paul, John) and (Paul, Sally).

Figure 15 illustrates the logical plan generated by BIGDANSING to clean this RDF dataset while the rest of Figure 14 explains the role of each logical operator on the RDF data. The plan starts with the Scope operator where it removes unwanted RDF projects on attributes Subject and Object and passes only those triples with *advised_by* and *study_in* predicates to the next operator. After that, we apply the first Block to group triples based on student name (*i.e.,* Paul) followed by Iterate operator to join the triples

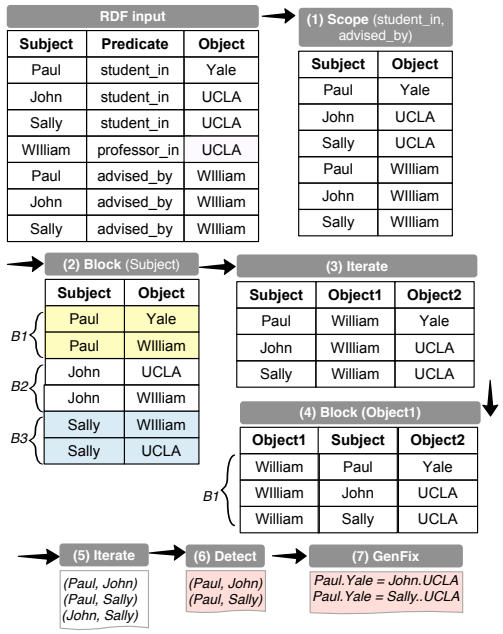


Figure 13: Logical operators execution for the RDF rule example.

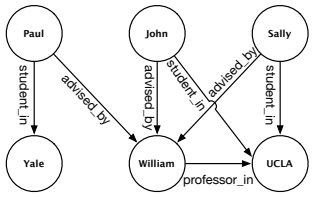


Figure 14: Example of an RDF graph.

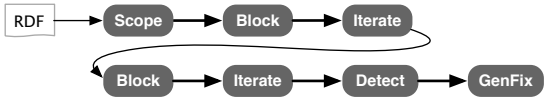


Figure 15: An RDF logical plan in BigDansing.

in each group. The output of this Iterate is passed to the second Block and Iterate operators to group incoming triples based on advisor name (*i.e.,* William). The Detect operator generates violations based on incoming triples with different university names. Finally, the GenFix operator suggests fixing the university names in incoming violations.

D. SPARK CODE FOR ϕ_F IN EXAMPLE 1

We show in Listing 7 an example of the physical translation of the logical plan for rule ϕ_F in Spark. Lines 3 to 7 in Listing 7 handles the processing of the UDF operators in Listings 4, 5, 6, 1 and 2. Lines 8 to 16 implement the repair algorithm in Section 5.2. The repair algorithm invokes Listing 8 in line 9 to build the graph of possible fixes and finds the graph's connected components through GraphX. Finally, lines 17 to 19 in Listing 7 applies the selected candidate fixes to the input dataset. Note that the Spark plan translation in Listing 7 is specific to rule ϕ_F, where lines 3 to 16 changes according the the input rule or UDF. The repair algorithm in Listing 7 seems far complex than the rule engine. However, as the number of violations in a dataset does not usually exceeds 10% of the input dataset size, the rule engine dominates the execution runtime because it has

```
//Input
JavaPairRDD<LongWritable,Text> inputData =
    sc.hadoopFile(InputPath,
    org.apache.hadoop.mapred.TextInputFormat.class,
    LongWritable.class, Text.class, minPartitions);
JavaRDD<Tuple> tupleData = tmpLogData.map(new
    StringToTuples());
//----Rule Engine----
JavaRDD<Tuple> scopeData = tupleData.map(new fdScope());
JavaPairRDD<Key,Iterable<Tupke>> blockData =
    scopedData.groupByKey(new
    fdBlock());
JavaRDD<TuplePair> iterateData = blockingData.map(new
    fdIterate());
JavaRDD<Violation> detectData = iterateData.map(new fdDetect());
JavaRDD<Fix> genFixData = detectData.map(new fdGenFix());
//----Repair Algorithm----
JavaRDD<Edge<Fix>> edges = genFixData.map(new extractEdges());
JavaRDD<EdgeTriplet<Object, Fix>> ccRDD = new
    RDDGraphBuilder(edges.rdd()).buildGraphRDD();
JavaPairRDD<Integer, Fix> groupedFix = ccRDD.mapToPair(new
    extractCC());
JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>>
    stringFixUniqueKeys = groupedFix .flatMapToPair(new
    extractStringFixUniqueKey());
JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>>
    countStringFixes = stringFixUniqueKeys .reduceByKey(new
        countFixes());
JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>>
    newUniqueKeysStringFix = countStringFixes .mapToPair(new
    extractReducedCellValuesKey());
JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>>
    reducedStringFixes = newUniqueKeysStringFix .reduceByKey(new
    reduceStringFixes());
JavaPairRDD<Integer,Fix> uniqueKeysFix =
    groupedFix.flatMapToPair(new
    extractFixUniqueKey());
JavaRDD<Fix> candidateFixes candidateFixes =
    uniqueKeysFix.join(reducedStringFixes).values().flatMap(new
        getFixValues());
//----Apply results to input----
JavaPairRDD<Long, Iterable<Fix>> fixRDD = candidateFixes.keyBy(
    new getFixTupleID()).groupByKey();
JavaPairRDD<Long, Tuple> dataRDD = tupleData.keyBy(new
    getTupleID());
JavaRDD<Tuple> newtupleData =
    dataRDD.leftOuterJoin(fixRDD).map(
    new ApplyFixes());
```

Listing 7: Spark code for rule ϕ_F

more data to process compared to the repair algorithm.

```
class RDDGraphBuilder(var edgeRDD: RDD[Edge[Fix]]) {
    def buildGraphRDD:JavaRDD[EdgeTriplet[VertexId, Fix]] = {
        var g: Graph[Integer, Fix] = Graph.fromEdges(edgeRDD, 0)
        new JavaRDD[EdgeTriplet[VertexId,Fix]](
            g.connectedComponents().triplets)
    }
}
```

Listing 8: Scala code for GraphX to find connected components of fixes

E. BUSHY PLAN EXAMPLE

We show a bushy plan example in Figure 16 based on the following two tables and DC rules from [6]:

Table Global (G): GID, FN, LN, Role, City, AC, ST, SAL

Table Local (L): LID, FN, LN, RNK, DO, Y, City, MID, SAL

$(c1) : \forall t_1, t_2 \in G, \neg(t_1.\text{City} = t_2.\text{City} \wedge t_1.\text{ST} \neq t_2.\text{ST})$

$(c2) : \forall t_1, t_2 \in G, \neg(t_1.\text{Role} = t_2.\text{Role} \wedge t_1.\text{City} = "NYC" \wedge t_2.\text{City} \neq "NYC" \wedge t_2.\text{SAL} > t_1.\text{SAL})$

$(c3) : \forall t_1, t_2 \in L, t_3 \in G, \neg(t_1.\text{LID} \neq t_2.\text{LID} \wedge t_1.\text{LID} = t_2.\text{MID} \wedge t_1.\text{FN} \approx t_3.\text{FN} \wedge t_1.\text{LN} \approx t_3.\text{LN} \wedge t_1.\text{City} = t_3.\text{City} \wedge t_3.\text{Role} \neq "M")$

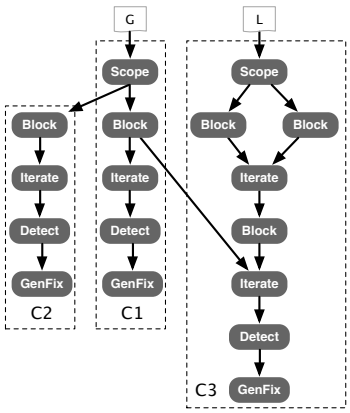


Figure 16: Example of bushy plan

The plan starts with applying the Scope operator. Instead of calling Scope for each rule, we only invoke Scope for each relation. Next we apply the Block operator as follows: block on "City" for c1, on "Role" for c2, and on "LID" and "MID" for c3. Thereafter, for c1 and c2, we proceed to iterate candidate tuples with violations (Iterate) and feed them to Detect and GenFix operators respectively. For c3, we iterate over all employees who are managers combine them with data units from the global table G and then finally feed them to the Detect and GenFix operators.

The key thing to note in the above bushy data cleaning plan is that while each rule has its own Detect/GenFix operator, the plan shares many of the other operators in order to reduce: (1) the number of times data is read from the base relations, and (2) the number of duplicate data units generated and processed in the dataflow.

F. DATA STORAGE MANAGER

BIGDANSING applies three different data storage optimizations: (i) data *partitioning* to avoid shuffling large amounts of data; (ii) data *replication* to efficiently support a large variety of data quality rules; and (iii) data *layouts* to improve I/O operations. We describe them below.

(1) Partitioning. Typically, distributed data storage systems split data files into smaller chunks based on size. In contrast, BIGDANSING partitions a dataset based on its content, *i.e.,* based on attribute values. Such a logical partitioning allows to co-locate data based on a given blocking key. As a result, BIGDANSING can push down the Block operator to the storage manager. This allows avoiding to co-locate datasets while detecting violations and hence to significantly reduce the network costs.

(2) Replication. A single data partitioning, however, might not be useful for mutliple data cleansing tasks. In practice, we may need to run several data cleansing jobs as data cleansing tasks do not share the same blocking key. To handle such a case, we replicate a dataset in a heterogenous manner. In other words, BIGDANSING logically partitions (*i.e.,* based on values) each replica on a different attribute. As a result, we can again push down the Block operator for multiple data cleansing tasks.

(3) Layout. BIGDANSING converts a dataset to binary format when storing it in the underlying data storage framework. This helps avoid expensive string parsing operations.

Also, in most cases, binary format ends up reducing the file size and hence I/Os. Additionally, we store a dataset in a column-oriented fashion. This enables pushing down the Scope operator to the storage manager and hence reduces I/O costs significantly.

As underlying data storage layer, we use Cartilage [21] to store data to and access data from HDFS. Cartilage works both with Hadoop and Spark and uses HDFS as the underlying distributed file system. Using Cartilage, the storage manager essentially translates BigDansing data access operations (including the operator pushdowns) to three basic HDFS data access operations: (i) *Path Filter*, to filter the input file paths; (ii) *Input Format*, to assign input files to workers; and (iii) *Record Reader*, to parse the files into tuples. In Spark, this means that we specify these three UDFs when creating RDDs. As a result, we can manipulate the data access right from HDFS so that this data access is invisible and completely non-invasive to Spark. To leverage all the data storage optimizations done by BigDansing, we indeed need to know how the data was uploaded in the first place, e.g., in which layout and sort order the data is stored. To allow BigDansing to know so, in addition to datasets, we store the *upload plan* of each uploaded dataset, which is essentially the upload metadata. At query time, BigDansing uses this metadata to decide how to access an input dataset, *e.g.,* if it performs a full scan or an index scan, using the right UDF (path filter, input format, record reader) implementation.

G. EXECUTION LAYER

G.1 Translation to Spark Execution Plans

Spark represents datasets as a set of *Resilient Distributed Datasets* (RDDs), where each RDD stores all Us of an input dataset in sequence. Thus, the Executor represents each physical operator as a RDD data transformation.

Spark-PScope. The Executor receives a set of RDDs as well as a set of PScope operators. It links each RDD with one or more PScope operators, according to their labels. Then, it simply translates each PScope to a map() Spark operation over its RDD. Spark, in turn, takes care of automatically parallelizing the map() operation over all input Us. As a PScope might output a null or empty U, the Executor applies a $filter()$ Spark operation to remove null and empty Us before passing them to the next operator.

Spark-PBlock. The Executor applies one Spark groupBy() operation for each PBlock operator over a single RDD. BigDansing automatically extracts the key from each U in parallel and passes it to Spark, which in turn uses the extracted key for its groupBy operation. As a result, Spark generates a RDDPair (a key-value pair data structure) containing each a grouping key (the key in the RDDPair) together with the list of all Us sharing the same key (the value in the RDDPair).

Spark-CoBlock. The Executor receives a set of RDDs and a set of PBlock operators with matching labels. Similar to the Spark-PBlock, the Spark-CoBlock groups each input RDD (with groupBy()) using its corresponding PBlock. In addition, it performs a $join()$ Spark operation on the keys of the output produced by groupBy(). Spark-CoBlock also outputs an RDDPair, but in contrast to Spark-PBlock, the produced value is a set of lists of Us from all input RDDs sharing the same extracted key.

Spark-CrossProduct & -UCrossProduct. The Executor receives two input RDDs and outputs an RDDPair of the resulting cross product. Notice that we extended Spark's Scala code with a new function $selfCartesian()$ in order to efficiently support the UCrossProduct operator. Basically, $selfCartesian()$ computes all the possible combinations of pair-tuples in the input RDDs.

Spark-OCJoin. The Executor receives two RDDs and a set of inequality join conditions as input. The Executor applies the OCJoin operator on top of Spark as follows. First, it extracts PartAtt (the attribute on which it has to partition the two input RDDs) from both RDDs by using the $keyBy()$ Spark function. Then, the Executor uses the $sortByKey()$ Spark function to perform a range partitioning of both RDDs. As a result, the Executor produces a single RDD containing several data blocks using the $mapPartitions()$ Spark function. Each data block provides as many lists as inequality join conditions; each containing all Us sorted on a different attribute involved in the join conditions. Finally, the Executor uses the $selfCartesian()$ Spark function to generate unique sets of paired data blocks.

Spark-PDetect. This operator receives a PIterate operator, a PDetect operator, and a single RDD as input. The Executor first applies the PIterate operator and then the PDetect operator on the output. The Executor implements this operator using the $map()$ Spark function.

Spark-PGenFix The Executor applies a PGenFix on each input RDD using spark's $map()$ function. When processing multiple rules on the same input dataset, the Executor generates an independent RDD of fixes for each rule. After that it combines all RDDs of possible repairs into a single RDD and pass it to BigDansing's repair algorithm. This operator is also implemented by the Executor inside the Detect operator for performance optimization purposes.

G.2 Translation to MR Execution Plans

We now briefly describe how the Executor runs the four wrapper physical operators on MapReduce.

MR-PScope. The Executor translates the PScope operator into a Map task whose *map* function applies the received PScope. Null and empty U are discarded within the same Map task before passing them to the next operator.

MR-PBlock. The Executor translates the PBlock operator into a Map task whose *partitioner* function applies the received PBlock to set the intermediate key. The MapReduce framework automatically groups all Us that share the same key. The Executor does the same for the CoBlock operator, but it also labels each intermediate key-value pair with the input dataset label for identification at Reduce tasks.

MR-PIterate. The Executor translates PIterate into a Reduce task whose *reduce* function applies the received PIterate.

MR-PDetect. The Executor translates the PDetect operator into a Reduce task whose *reduce* function applies the received PDetect. The Executor might also apply the received PDetect in the *reduce* function of a Combine task.

MR-PGenFix. The Executor translates the PGenFix operator into a Map task whose *map* function applies the received PRepair. The Executor might also apply the received PGenFix at the *reduce* function of PDetect.