CleanM: An Optimizable Query Language for Unified Scale-Out Data Cleaning

Stella Giannakopoulou[†] Manos Karpathiotakis[†] Benjamin Gaidioz[†] Anastasia Ailamaki[†‡]

[†]Ecole Polytechnique Fédérale de Lausanne
firstname.lastname@epfl.ch

[‡]RAW Labs SA

ABSTRACT

Data cleaning has become an indispensable part of data analysis due to the increasing amount of dirty data. Data scientists spend most of their time preparing dirty data before it can be used for data analysis. At the same time, the existing tools that attempt to automate the data cleaning procedure typically focus on a specific use case and operation. Still, even such specialized tools exhibit long running times or fail to process large datasets. Therefore, from a user's perspective, one is forced to use a different, potentially inefficient tool for each category of errors.

This paper addresses the coverage and efficiency problems of data cleaning. It introduces CleanM (*pronounced clean'em*), a language which can express multiple types of cleaning operations. CleanM goes through a three-level translation process for optimization purposes; a different family of optimizations is applied in each abstraction level. Thus, CleanM can express complex data cleaning tasks, optimize them in a unified way, and deploy them in a scaleout fashion. We validate the applicability of CleanM by using it on top of CleanDB, a newly designed and implemented framework which can query heterogeneous data. When compared to existing data cleaning solutions, CleanDB a) covers more data corruption cases, b) scales better, and can handle cases for which its competitors are unable to terminate, and c) uses a single interface for querying and for data cleaning.

1. INTRODUCTION

Today's ever-increasing rate of data volume and variety opens multiple opportunities; crawling through large-scale datasets and analyzing them together reveals data patterns and actionable insights to data analysts. However, the process of gathering, storing, and integrating diverse datasets introduces several inaccuracies in the data: Analysts spend 50%-80% of their time preparing dirty data before it can be used for information extraction [34]. Therefore, data cleaning is a major hurdle for data analysis.

Data cleaning is challenging because errors arise in different forms: Syntactic errors involve violations such as values out of domain or range. Semantic errors are also frequent in non-curated datasets; they involve values which are seemingly correct, e.g., Beijing is located in the US. In addition, the presence of duplicate entries is a typical issue when integrating multiple data sources. Besides requiring accurate error detection and repair, the aforementioned data cleaning tasks also involve computationally intensive operations such as inequality joins, similarity joins, and multiple scans of each involved dataset. Thus, it is difficult to build general-purpose tools that can capture the majority of error types and at the same time perform data cleaning in a scalable manner.

Existing data cleaning approaches can be classified into two main categories: The first category includes interactive tools through which a user specifies constraints for the columns of a dataset or provides example transformations [25, 38]. User involvement in the cleaning process is intuitive and interactive, yet specifying all possible errors involves significant manual effort, especially if a dataset contains a large number of discrepancies. The second category comprises semi-automatic tools which enable several data cleaning operations [14, 19, 29, 43]. Both categories lack a universal representation for users to express different cleaning scripts, and/or are unable to optimize different cleaning operations as one unified task because they treat each operation as a black-box UDF.

Therefore, there is need for a higher-level representation for data cleaning that serves a purpose similar to that of SQL for data management in terms of expressivity and optimization: First, SQL allows users to manage data in an organized way and is subjective to how each user wants to manipulate the data. Similarly, data cleaning is a task which is subjective to the user's perception of cleanliness and therefore requires a language that allows users to express their requests in a simple yet efficient way. Second, SQL is backed by the highly optimizable relational calculus; data cleaning tasks require an optimizable underlying representation as well.

This paper introduces CleanM, a declarative query language for expressing data cleaning tasks. Based on SQL, CleanM offers primitives for all popular cleaning operations and can be extended to express more operations in a straightforward way. CleanM follows a three-level optimization process; each level uses a different abstraction to better suit the optimizations to be applied. First, all cleaning tasks expressed using CleanM are translated to the *monoid comprehension calculus* [18]. The monoid calculus is an optimizable calculus which is inherently parallelizable and can also represent complex operations between various data collection types. Then, comprehensions are translated into an intermediate algebra which allows for inter-operator optimizations and detection of work sharing opportunities. Finally, the algebraic operators are translated into a physical plan which is then optimized for factors such as data skew. In summary, regardless of how complex a cleaning task is, whether it internally invokes complex operations such as clustering, and what the underlying data representation is (relational,

Proceedings of the VLDB Endowment, Vol. 10, No. 11

JSON, etc.), the overall task will be treated as a single query, optimized as a whole, and executed in a distributed, scale-out fashion.

We validate CleanM by building CleanDB, a distributed data cleaning framework. CleanDB couples Spark with a CleanM frontend and with a cleaning-oriented optimizer which applies the three-level optimization process described above. The end result is a system that combines data cleaning and querying, all while relying on optimizer rewrites and abundant parallelism to speed up execution.

Motivating Example. Consider a dataset comprising customer information. Suppose that a user wants to validate customer names based on a dictionary, check for duplicate entries, and at the same time check whether a functional dependency holds. We will be using this compound cleaning task to reflect the capabilities of CleanM and CleanDB: For example, CleanM enables name validation via token filtering [24] – a common clustering-based data cleaning operation – by representing it as a monoid. Also, CleanDB identifies a rewriting opportunity to merge the duplicate elimination and functional dependency checks in one step.

Contributions: Our contributions are as follows:

- We introduce CleanM, an all-purpose data cleaning query language. CleanM models both straightforward cleaning operations such as syntactic checks, as well as complex cleaning building blocks such as clustering algorithms, all while being naturally extensible and parallelizable. We also present a three-level optimization process that ensures that a query expressed in CleanM results in an efficient distributed query plan.
- We implement CleanDB, a scale-out data cleaning framework that serves as a testbed for users to try CleanM. CleanDB supports a multitude of data cleaning operations (e.g., duplicate elimination, denial constraint checks, term validation) over multiple different types of data sources (e.g., binary, CSV, JSON, XML), executed in a distributed fashion using the Spark runtime.
- We show that CleanDB outperforms state-of-the-art solutions in synthetic and real-world workloads. CleanDB scales better than Spark SQL [6] and a dedicated scale-out data cleaning solution, offers a wider variety of operations, and cleans datasets that its competitors are unable to process due to performance issues.

In summary, current data cleaning technology lacks a universal representation that can be general and also guarantee scalability out-of-the-box for all the cleaning operations it supports. This paper provides a solution through an algebraic abstraction, which allows rich features to be embedded in a declarative, optimizable, and parallelizable language. The user can thus intertwine analytics and cleaning using a unified interface over a scale-out system.

2. RELATED WORK

This section surveys related work ([3, 11, 21, 23, 25, 29, 38, 43, 44]) and highlights how CleanM pushes the state of the art further.

Interactive Data Cleaning. Potter's Wheel [38], OpenRefine [44], and Trifacta – the commercial version of Data Wrangler [25] – are established interactive data cleaning systems. Potter's Wheel [38] provides an interface via which the user gradually repairs her dataset. The user performs transformations, such as merging columns, and at the same time, a background daemon detects potential syntactic errors. For the daemon to detect any errors, a user has to specify a set of domains to which data entries must belong, and the constraints of each domain. Knime [1] allows for more complex operations, but relies on black box UDFs. CleanM and CleanDB opt for a declarative approach to data cleaning compared to the manual "cleaning by example" of the previous systems, and also expose a superset of their functionality.

(Semi-)Automatic Cleaning. Besides interactive cleaning toolkits, other systems attempt to detect and repair data errors automatically, asking a human for guidance when necessary.

DataXFormer [3] tackles semantic transformations, such as mapping a column containing company names to a column with the stock symbols of those companies, by exploiting information from the Web or from mapping tables. Tamr [43] resolves data duplicates; it relies on training classifiers and feedback from experts to make deduplication suggestions. Dedoop [31] allows specifying entity resolution workflows through a web-based interface and then translates them into MapReduce jobs. Contrary to the unified optimization process of CleanM, each Dedoop operator is a standalone, black-box UDF. SampleClean [46] and Wisteria [22] extract a sample out of a dataset, employ users to clean it, and utilize this sample to answer aggregate queries; their focus is on data transformations, deduplication, and denial constraints – a subset of CleanM .

NADEEF [12, 14] manages a set of denial constraints, and tries to update erroneous values in a way that all the rules are satisfied [12]. BigDansing [29] ports the insights of NADEEF in a distributed setting by extending MapReduce-like frameworks with support for duplicate elimination and denial constraints. BigDansing's logical-level optimizations focus on projection push down and on grouping the tuples based on a specific attribute in order to prune the number of required comparisons. BigDansing also employs physical-level optimizations, such as a custom join implementation. CleanDB incorporates such physical and logical optimizations, but also allows for optimizations in all levels of the query translation process, such as language level simplification of expressions and coalescing of different operations.

Declarative Cleaning. The `FUSE BY` [9] operator is a proposed extension of SQL that resolves duplicates by allowing various conflict resolution strategies, such as choosing the most common value or preferring one source over another. `FRAQL` [40] follows a similar approach by providing SQL extensions that allow transformations, duplicate elimination, and outlier detection. All conflict resolution operations in `FRAQL` are expressed in the form of standalone, opaque UDFs. QuERy [5] integrates deduplication with query processing by focusing on the optimizations that allow cleaning only the parts of the data that are needed by a given query. Therefore, QuERy addresses a different dimension of the scalability issue of data cleaning than CleanM does, i.e., it avoids cleaning the whole dataset. Ajax [19] separates the logical and physical level of the data cleaning process. At the logical level, Ajax uses a data flow graph to represent the steps of a cleaning operation. Then, at the physical level, each logical operator gets translated into an optimized implementation. Like `FRAQL`, Ajax provides a UDF for each operator, and therefore treats each data cleaning task as a black box. On the contrary, CleanM comprises composable operations, which it optimizes both on their own and as a whole.

Quantitative Data Cleaning (QDC). QDC [7, 15] discovers the best data repairing strategy using statistical methods, such as the cost of each strategy, the quality of the resulting dataset, and the statistical distortion against the original dataset. QDC differs from CleanM in that it focuses on discovering the optimal repair method given a set of detected errors, whereas CleanM focuses on the detection of the errors. Statistics are also employed to measure the accuracy of error detection methods and how each method behaves in the existence of multiple types of errors; whether a method fails to detect an error due to the presence of another type of error [8].

SQL for cleaning. SQL can express some cleaning tasks, e.g., the ones that correspond to first order logic statements [16]. SQL, however, is overall inappropriate and insufficient for data cleaning: First, SQL lacks first-class support for rich data types (e.g., JSON);

one might need to convert a dataset to another format in order to clean it. A change in the intended format can be inconvenient for the user, or might complicate the cleaning process, e.g., flattening a dataset can increase data volume. In addition, relational algebra – the backend of SQL – lacks first-class support for operations from the machine learning and data mining domains.

It typically takes a combination of vanilla SQL, UDFs, extra operators, and external programs to express rich operations in SQL [13]. UDFs, however, increase complexity; each UDF appears as a black-box to the system optimizer, which is unable to optimize the entire task as a whole. Adding extra operators in the database core [37] requires coding in an operator per algorithm, which is a tedious process. As for frameworks such as Spark [48], which support both relational and iterative processing, they apply only relational optimizations [6]. The reason is that the "relational part" of Spark is engineered similarly to a DBMS with columnar storage and is equipped with an optimizer, whereas the "procedural part" executes arbitrary code over BLOB-like data (RDDs [48]). Given the split Spark architecture, the Spark SQL Catalyst optimizer treats the procedural parts of an analysis script as black boxes. In summary, both for traditional RDBMS and modern scale-out frameworks, while a relational optimizer can perform rewrites based on the physical properties of the extra operators, it is non-trivial to reason about them on an algebraic level, because they fall outside of the relational logic based on which the system has been engineered. Our running example highlights two ways in which systems engineered based on "vanilla SQL" are unsuitable for data cleaning: First, the term validation operation creates bags of values, which RDBMS typically treat as BLOB-like opaque data types, thus hurting performance. Second, an RDBMS query would treat each of the three cleaning operations as standalone; as we will see in Section 5, however, two of these operations can share work.

In conclusion, SQL is designed to manipulate relational data, and is unable to express domain-specific optimizations required for data cleaning. On the contrary, CleanM is specifically designed to express complex cleaning operations over complex data types.

3. A UNIFIED REPRESENTATION

Data cleaning is a computationally intensive process which typically involves multiple iterations over the same dataset and numerous pairwise comparisons of the data records. In fact, many data cleaning tasks would benefit from machine learning operations such as clustering in order to split a dataset into manageable subsets and minimize the number of required pairwise comparisons. Therefore, a data cleaning language must be coupled with a calculus that can support and optimize such operations. At the same time, said calculus must be able to reason about multiple cleaning operations as a whole, and identify inter- and intra-operation optimizations. Besides involving complex operations, data cleaning tasks are typically applied over a variety of data sources and formats. Data that requires curation may be i) relational or not, ii) stored in a DBMS or kept in files [4, 27], etc. Therefore, a data cleaning language and calculus must be able to handle data heterogeneity. Finally, given the ever-increasing data volumes, explicit support of parallelism is a prerequisite. This section presents i) the cleaning operations that CleanM supports, and ii) the rationale behind a three-level translation of said cleaning operations into executable code.

3.1 Data cleaning operations

In the following we present the data cleaning operations that CleanM supports, and what is required to optimize each operation.

Denial Constraints (DC)

The family of *denial constraints* [16] contains universally quantified first order language sentences that represent data dependencies, such as functional dependencies and conditional functional dependencies. DCs have the following form: $\forall t_1,...,t_k \neg(p(x_1) \wedge p(x_2) \wedge ...p(x_n))$. If a dataset contains one or more tuples for which the predicates $p(x_1)...p(x_n)$ hold, it is considered to be inconsistent.

Optimization Requirements. DC checks involve a selection or a self-join that detects tuples, pairs of tuples, or groups of tuples that violate the rule. Self-joins are expensive because they involve multiple traversals of the input. Also, as DCs contain arbitrary predicates, such as inequalities, *theta*-joins might be required. Finally, the rules need to handle non-exact matches, and thus similarity joins are also required. Similarity joins are costly operations because they involve multiple passes over a dataset, as well as a computationally expensive similarity check per candidate pair.

Duplicate Elimination

Duplicate elimination involves the discovery of tuples that refer to the same real-world entity [32]. The most straightforward way to detect similar tuples is a self-join that discovers identical tuples. A lighter duplicate detection form is to consider an attribute or a set of attributes that should be unique; if two tuples have the same values for that particular set of attributes, then they are considered to be duplicates. A more challenging scenario involves the case where a dataset does not contain completely identical pairs of tuples/attribute sets, but might contain *similar* pairs. In this case, the self-join predicate needs to calculate similarity instead of equality.

Optimization Requirements. Similar to a subset of denial constraints, deduplication involves a similarity self-join to identify potentially duplicate records [23].

Transformations & Term Validation

Transformations involve applying a formula to a set of values, or mapping values to a set of semantically related values [3]. Semantic transformations are challenging because they require consulting auxiliary data. *Term validation* is a popular category of semantic transformations: It focuses on detecting values that are seemingly correct, but fail to adhere to a specific terminology because of, for example, a misspelling. A common technique for detecting misspellings is using a dictionary for validation. The dictionary can be, among others, a dictionary of english words or scientific terms.

Optimization Requirements. Semantic transformations involve an *equi*-join or a similarity join with auxiliary data. Specifically, term validation requires the discovery of the most similar words from the dictionary for each word of the dataset. Thus, term validation relies on the efficient computation of similarity checks.

Summary. After surveying a range of data cleaning operations, we identify that efficient handling of self-, theta-, and similarity joins can accelerate multiple cleaning tasks. Besides accelerating standalone operations, having a unified representation for all operations can help in detecting common patterns and work sharing opportunities. Finally, having a principled way to simplify an arbitrary data cleaning script (e.g., unnest nested sub-tasks) makes detection of optimization opportunities over the script more straightforward.

3.2 From data cleaning operations to code

This work uses three different abstraction levels to reason about and optimize data cleaning tasks. In the first level, CleanM maps cleaning operations to the *monoid comprehension calculus* [18]. As a result, the operations are first-class citizens of the language instead of black-box UDFs. Such composability means that operations can be explicitly used and stacked with each other in monoid

comprehensions. Transforming the input dataset between different types and manipulating multiple data types is also possible, a feature exploited by engines that access raw data [26, 28]. Monoid comprehensions are inherently parallelizable and lend themselves perfectly to scale-out execution – a fact that has led existing scale-out approaches to adapt monoids as a core abstraction for data aggregation and incremental query processing [10, 17]. Section 4 elaborates on how cleaning operations are mapped to CleanM.

The second abstraction level involves lowering a comprehension into an algebraic form [18], the *nested relational algebra*. Nested relational algebra operators resemble relational operators and are amenable to relational-like optimizations, yet they also explicitly handle complex data types and queries. For example, a user can issue a query combining relational and hierarchical data, and rely on the algebraic translation process to simplify the physical query plan and remove all forms of query nesting. In addition, the algebraic form enables inter-operator rewrites, which coalesce different cleaning operations into a single one and thus reduce the overall cost. Section 5 discusses the algebraic rewrites.

The final level specializes the algebraic expression to the underlying execution engine. CleanM currently assumes that Spark [48] is the underlying engine; still, it is pluggable to any scale-out system. This physical level focuses on the particularities of cleaning operations such as the presence of expensive theta joins. Also, the physical level addresses the absence of uniform distribution in the values of real-world datasets – a fact that can cause load imbalance during data cleaning. Section 6 discusses how to generate physical plans that consider both these complications.

4. CLEANING DATA USING MONOIDS

CleanM supports multiple cleaning operations, which it internally maps to monoid comprehensions. Still, although a unified representation is important for user convenience, it is also important to optimize each of the operations. In addition, despite the elegance of comprehensions, the goal of CleanM is to serve as a SQL-like higher-level representation that masks the comprehension syntax, given that most users are more familiar with SQL. The syntax of CleanM extends SQL with constructs that express data cleaning operations and handle non-relational data types such as hierarchies; this work focuses on the data cleaning operations.

This section presents i) monoid comprehensions (the underlying calculus of CleanM), ii) the optimizations that comprehensions allow, iii) the expressive power of CleanM by showing how to map the building blocks of data cleaning operations to monoids, and iv) the syntax and semantics of CleanM .

4.1 The monoid comprehension calculus

CleanM translates data cleaning tasks into expressions of the monoid comprehension calculus. A *monoid* is an algebraic structure which captures aggregate and collection operators. A primitive monoid m models aggregate operators. It is accompanied by an associative *merge* operation $\oplus$ and an identity/zero element $\mathbb{Z}_{\oplus}$. For example, the max operation over positive integers corresponds to the monoid $(max, 0)$, because computing the max is an associative operation, with 0 being its zero element. A collection monoid comprises a merge operation, a zero element, and a unit function $\mathcal{U}_{\oplus}$ to construct singleton values of a monoid type. For example, a list collection can be modeled as a monoid, because the list append operation $++$ is associative, the empty list $[]$ is its zero element, and the function $a \rightarrow [a]$ is its unit function.

A *monoid comprehension* of the form $\oplus\{e|q_1,...,q_n\}, n \geq 0$ describes operations between monoids. $q_1,...,q_n$ form the comprehension *body*; each of them is either a filter predicate, or a generator of the form $var \leftarrow col$ that iterates through an instance of a monoid collection type, and assigns the currently visited element to a variable. e is the head of the comprehension, indicating the expression to be computed for every value combination of the bound variables. Finally, the $\oplus$ symbol is the merge operation of the output monoid, indicating how to aggregate the instantiations of e.

Example. The comprehension $+\{x|x \leftarrow [1,2,10], x < 5\}$ computes the *sum* of the elements that are smaller than 5 for a given list, and the comprehension $set\{(x,y)|x \leftarrow \{1,2\}, y \leftarrow \{3,4\}\}$ produces the cross product of two data collections. This paper uses Scala-like comprehension syntax which is equivalent to the one presented, representing a comprehension as $for\{q_1,...,q_n\}\ yield\ \oplus\ e$.

4.2 Optimizations at the monoid level

As discussed in Section 3.2, CleanM follows a layered design approach. Even in its topmost layer, CleanM distinguishes between high- and low-level operations, both of which are first-class citizens and are expressed using comprehensions. The separation aims for user convenience: High-level operations, such as denial constraints, map directly to a SQL-like, syntactic sugar representation. Low-level operations are internal building blocks for the high-level ones, and address the optimization requirements of Section 3.1. Both high- and low-level operations go through a rewrite process that applies general-purpose, domain-agnostic optimizations [18].

Domain-agnostic optimizations: Normalization

Regardless of the processing that a comprehension performs, a normalization algorithm [18] puts it into a "canonical" form. The normalization also applies a series of optimization rewrites. Specifically, it applies filter pushdown and operator fusion. In addition, it flattens multiple types of nested comprehensions [30]. It also replaces any function call that appears in a comprehension, with the call's result (*beta reduction*); a function's input can be an arbitrary expression (e.g., a constant, a generator's variable, etc.). In the case of UDFs that are defined as comprehensions themselves, the rewrite results in their unnesting, and facilitates optimizing the rewritten comprehension as a whole. Also, it splits if-then-else expressions in two comprehensions, so that each one of them can be further optimized. Similar to the SQL-based rewriting of the EXISTS clause, normalization unnests existential quantifications. Finally, normalization simplifies expressions that are statically known to evaluate to true/false, and presences of empty collections.

The result of the normalization process is a simplified comprehension; Section 5 explains how this comprehension is further rewritten into a form more suitable for efficient execution.

Domain-specific optimizations: Pruning comparisons

Besides domain-agnostic optimizations, the monoid calculus can express operations that specifically target and accelerate data cleaning tasks. A common theme of all the data cleaning operations mentioned in Section 3.1 is the need for fast pairwise comparisons. The rest of this section discusses how to optimize CleanM expressions on the comprehension level by pruning comparisons in the cases of self-joins and similarity joins; we discuss the rest of the optimization requirements of Section 3.1 in subsequent sections because they are a better match for lower abstraction levels.

Self-joins occur in denial constraints (DC) and duplicate elimination. In the case of self-joins that involve equality conditions, such as in functional dependencies (FD), CleanM avoids the self-join by grouping the dataset's entries based on the left hand side of the FD, and then detects violations (i.e., whether a grouping key is associated with more than one value). Section 6 discusses how

CleanM handles the general case of DCs, which may involve non-equality predicates, in its third abstraction level – the physical one.

Regarding similarity joins, a baseline method to evaluate them would compute the cartesian product and afterwards apply a filter that removes the dissimilar pairs. The baseline approach, however, is very costly, because both the cartesian product and the string similarity computation are expensive tasks. Thus, CleanM uses a filtering phase to prune the candidate pairs that need to be checked. An indicative example of filtering is the use of a clustering algorithm to create k clusters, each containing words that are similar. Then, the cleaning operation only has to perform intra-cluster comparisons. The pre-processing filtering phase must be lightweight enough to avoid adding an overhead that reaches the cost of an unoptimized implementation. Thus, CleanM considers variations of the approaches suggested in [24, 39], namely k-means and token filtering, because different clustering/filtering techniques are more suitable for different use cases; their efficiency in the context of data cleaning depends on several factors, such as the string length of a dataset's words and the similarity metric used. Still, to use any technique, we must be able to express it as a monoid.

4.3 Expressive Power: Mapping cleaning building blocks to the monoid calculus

Expressing an operation over type T as a monoid involves either mapping the operation to an existing monoid, or proving three properties: First, specifying an identity/zero element $\mathbb{Z}_\oplus$ such that for any element x of type T, $x \oplus \mathbb{Z}_\oplus = \mathbb{Z}_\oplus \oplus x = x$. Second, specifying a unit function that turns an element into a singleton value of T. Third, showing that the associative property $\oplus$ holds for it. Multiple operations over collections such as lists, bags, sets, arrays, vectors, etc., are provably mappable to the monoid calculus [18]. Also, monoid comprehensions are sufficient to represent OQL and SQL queries [18]. The rest of this section elaborates on how to map clustering and filtering algorithms – which CleanM relies on to refine similarity joins – to the monoid calculus.

Clustering as a monoid

Clustering algorithms can be divided into partitional and hierarchical. Below, we map each category to the monoid calculus.

Single-pass partitional algorithms. Partitional algorithms split the input into a number of clusters. Each element of the dataset might belong to exactly one (strict) or more clusters (overlapping). The assignment of a value to a cluster depends on certain criteria, such as the distance from the cluster center (k-means) or the distance from the other elements of the cluster (DBSCAN). In the following, we provide the mapping of k-means – the most popular partitional algorithm – to the monoid calculus; mapping other partitional algorithms to the monoid calculus is straightforward by mapping different cluster assignment criteria.

K-means assigns each input element to the cluster which contains values that are similar to it; thus, when used in the context of similarity joins, only intra-cluster comparisons take place. CleanM by default uses a variation of k-means inspired by ClusterJoin [39]. The k-means variation selects k random centers, and then assigns each word of the dataset to all centers whose distance is minimum (or minimum plus a $delta$ to favor multiple assignments). The original k-means requires multiple iterations before converging to an optimal set of clusters, which hurts scalability. The k-means variation avoids scalability issues by only iterating once over the input, while also achieving a "good-enough" grouping of similar words.

Mapping the k-means single-pass operation over bag collections to the monoid calculus requires expressing the *center initialization* and the *center assignment* steps as monoid operations; the latter step is the one performing the actual clustering/partitioning.

We express *center initialization* by parameterizing *the function composition monoid* [18] instead of defining a new monoid. The function composition monoid can compose functions that propagate a state during an iteration over a collection, as long as the composed functions are associative. The "propagated state" at the end of the iteration comprises the centers for k-means. We parameterize the function composition monoid to apply randomized algorithms, such as reservoir sampling [45], to extract k centers. A straightforward parameterization is: $\circ\{\lambda(x,i).(if\ i = N/k,\ 2N/k,\ ...,\ N,\ then\ [x]++y, i-1) | y \leftarrow Y\}$. The formula extracts the $N/k, 2N/k, ..., N$ items as centers. x accumulates the result, and the initial value of i is the length of the original list. Extracting items using a fixed step is an associative operation because it appends specific elements to a bag collection in each iteration, thus the overall parameterization of the composition monoid is a monoid operation too.

Center assignment takes as a parameter the list of centers computed in the first step and discovers the closest center for each data item. This operation maps to the *Min* monoid [18].

Multi-pass partitional algorithms. Representing multi-pass partitional algorithms (e.g., the original k-means, canopy clustering [35], correlation clustering [42], etc.) as monoids is straightforward: The representation of iterative clustering algorithms implies n equivalent monoid comprehensions, where n is the number of iterations. Each iteration stores the result of the comprehension into a state which is then transferred to the next iteration. Alternatively, an *iteration monoid* can act as syntactic sugar in place of the n comprehensions; its behavior will resemble *foldLeft*, and it will update some state in each iteration.

Hierarchical clustering. Hierarchical clustering generates clusters that can have sub-clusters. Executing hierarchical clustering involves a set of iterations which gradually build the resulting clusters by merging or splitting items. In the monoid representation of hierarchical clustering, each iteration gets as input the previous state or the initial dataset, and computes the items whose distance from each other is minimum; this operation maps to the *Min* monoid.

(Token) filtering as a monoid

Token filtering [24] is the preferred way to reduce the number of similarity comparisons when comparing strings of small length, whereas clustering-based filtering is suitable for more generic use cases. Token filtering groups the words based on their tokens in order to avoid comparing all pairs exhaustively. Specifically, token filtering splits each word into tokens of length q, and then associates each token with the groups of words that contain said token. Thus, similarity checks only take place within each group.

The monoid representation of token filtering resembles that of k-means, in that k-means groups values based on their common "center", whereas token filtering groups them based on a common token. Below, we provide the mapping of token filtering into the monoid calculus. $[str_i, str_j, str_k]$ denotes that at least one of the three strings will be part of the set of values that contain the token.

$$\mathbb{Z}_\oplus : \{\},\ Unit : str \rightarrow \{(token_i, \{str\}), (token_j, \{str\})...\}$$

$$Associative\ property : tokenize(str_i, tokenize(str_j, str_k)) =$$

$$\{(token_i, \{[str_i, str_j, str_k]\}), (token_j, \{[str_i, str_j, str_k]\})...\} =$$

$$tokenize(tokenize(str_i, str_j), str_k)$$

Extensibility and scope of CleanM

Extending CleanM with any operation that obeys the monoid properties is straightforward. Besides k-means clustering and token filtering, CleanM can represent any filtering approach that groups

```
SELECT [ALL|DISTINCT] <SELECTLIST> <FROMCLAUSE>
[WHERECLAUSE][GBCLAUSE[HCLAUSE]][FD|DEDUP|CLUSTER BY]*

FD=FD(attributesLHS, attributesRHS)
DEDUP=DEDUP(<op>[,<metric>, <theta>][,<attributes>])
CLUSTERBY=CLUSTER BY(<op>[,<metric>,<theta>],<term>)
```

Listing 1: The syntax of CleanM.

words into clusters of similar contents (e.g., filtering based on the length of the words). Other filtering approaches such as applying transitive closure in order to build the similar pairs can be also represented using the monoid calculus.

Future work includes examining operations which lack an associative property (e.g., median), and which have traditionally been handled by scale-out systems via exponential algorithms or approximation. Finally, this work focuses on violation detection with minimal user effort; cleaning-oriented topics such as i) data repairing techniques and ii) techniques that rely on classification using an offline training phase and pre-existing training data are orthogonal extensions to our declarative language proposal.

4.4 The CleanM language

Having defined the necessary low-level operations, we describe the high-level cleaning operations of CleanM. CleanM extends SQL with data cleaning operators; its syntax is shown in Listing 1. The symbols ([]), (*) and (|) denote optional elements, elements that can appear multiple times, and option between elements, respectively. The symbol (|) implies arbitrary order between the options. When multiple cleaning operations appear in the CleanM query, then the semantics of the query correspond to an outer join that takes as input the violations of each cleaning operator that appears in the query, and outputs the entities that contain at least one violation. Except for the `[FD|DEDUP|CLUSTER BY]` part, the syntax and semantics of the operators are equivalent to that of SQL.

We now analyze the syntax of each cleaning operator and present the semantics of CleanM using the monoid calculus. We also go through the motivating example of the introduction, which checks the rule $address \rightarrow prefix(phone)$, detects duplicate customers using Levenshtein distance (LD) as similarity metric, and validates customer names using token filtering and a dictionary. The corresponding CleanM query is the following:

```
SELECT c.name,c.address, *
FROM customer c, dictionary d
FD(c.address, prefix(c.phone))
DEDUP(token filtering, LD, 0.8, c.address)
CLUSTER BY(token filtering, LD, 0.8, c.name)
```

Denial Constraints. The general category of denial constraints is expressible using vanilla SQL, thus CleanM reuses SQL syntax to express them. CleanM makes an exception for functional dependencies – the most popular sub-category of denial constraints – and uses the FD operator shown in Listing 1. The query result contains the entities that violate the FD rule. *LHS* and *RHS* correspond to the left and right-hand side of the rule. Both *LHS* and *RHS* can involve more than one attribute. The semantics of the `FD` operator correspond to the following comprehension:

```
groups:=for(d<-data) yield filter(d.term,algo),
for(g<-groups,g.count>1) yield bag g
```

The comprehension groups the input dataset using the *filter* monoid based on a *term* attribute and then returns the groups containing more than one item. The FD: $address \rightarrow prefix(phone)$ of the running example corresponds to the following comprehension:

```
groups:=for(c<-cust)yield filter(prefix(c.phone)),
for(g<-groups,g.count>1) yield bag g
```

Duplicate Elimination. The `DEDUP` operator of Listing 1 comprises the <op> field that represents the filtering operation to use for the similarity join, <metric>, which is the distance metric to be used (e.g., Jaccard, Euclidean), and <theta>, which is the similarity threshold. The <attributes> field represents the set of attributes that determine whether two entities are equal. <attributes>, <metric> and <theta> are optional – a default value is set if they are missing.

The query result contains the duplicate entities. The semantics of the `DEDUP` operator correspond to the following comprehension:

```
groups:=for(d <- data) yield filter(d.terms,algo),
for(g<-groups,p1<-g.partition,p2<-g.partition),
    similar(metric,p1.atts,p2.atts,θ))
yield bag(p1, p2)
```

The *filter* monoid groups the data based on the specified attributes or by building clusters based on that attributes. Then, the entries within each group are compared against each other using a similarity metric. The comprehension outputs pairs of records that are potential duplicates. `partition` is a built-in field that represents the set of records that correspond to each group. The comprehension of the deduplication part of the running example is the following:

```
groups:=for(c<-cust) yield filter(c.address,tf),
for(g<-groups,p1<-g.partition,p2<-g.partition),
LD(p1.atts,p2.atts)>0.8) yield bag(p1, p2)
```

Term Validation. The CleanM syntax for term validation requires the `CLUSTER BY` operator of Listing 1, which resembles `DEDUP`. The <term> field stands for the attribute(s) based on which the similarity is measured. `CLUSTER BY` requires also an additional table in the <FROMCLAUSE> that represents the dictionary.

The query result couples each dirty term with the set of dictionary terms that are similar to it. The similar dictionary terms correspond to the suggested repair of the invalid term. The semantics of `CLUSTER BY` correspond to the following comprehension:

```
dataGroup:=for(d<-data) yield filter(d.term,algo),
dictGroup:=for(d<-dict) yield filter(d.term,algo),
similarTerms:=for(d1<-dataGroup, d2<-dictGroup,
                  d1.key = d2.key,
                similar(metric,d1.term,d2.term,θ))
              yield list(d1.term, d2.term)
```

First, the input is clustered based on a *term* attribute whose values potentially contain inconsistencies. The same process is followed for the entries of the dictionary. Then, the comprehension tries to find similar data-dictionary pairs by comparing only the clusters that correspond to the same grouping key. The respective validation of the customer name in the running example is the following:

```
dataGroup:=for(c<-cust) yield filter(c.name,tf),
dictGroup:=for(d<-dict) yield filter(d.name,tf),
similarTerms:=for(d1<-dataGroup, d2<-dictGroup,
d1.key = d2.key,LD(d1.name,d2.name)>0.8)
yield list(d1.name, d2.name)
```

Transformations. CleanM differentiates between syntactic and semantic transformations. Syntactic transformations are lightweight repair operations such as splitting an attribute, and thus can be expressed using vanilla SQL. Semantic transformations require an auxiliary table that contains value mappings. Thus, they reuse the term validation constructs, with the difference that the projection list contains the desirable attribute from the auxiliary table as a suggested repair. For example, one could map airports to cities using an auxiliary table that contains airport-to-city mappings.

Summary. CleanM exposes users to a SQL-like extension: Each operator extends the syntax of SQL based on the functionality it resembles. Every operator is deeply integrated in CleanM instead of

Operator Name	Select	(Outer) Join	Reduce	Nest	(Outer) Unnest
Operator Symbol	$\sigma_p(X)$	$X \ltimes_p Y$ $X \bowtie_p Y$	$\Delta_p^{\oplus/e}$	$\Gamma_p^{\oplus/e/f}$	$\not\mu_p^{path}(X)$ $\mu_p^{path}(X)$
Superscript & Subscript	p: Filtering Expression f: Groupby Expression $\oplus$: Output Type / Monoid			e: Output Expression $path$: Field to unnest	

Table 1: The operators of the nested relational algebra.

being treated as a black-box UDF; all operators end up translated to the monoid comprehension calculus. Thus, CleanM treats cleaning operations as inherently parallelizable, offers operation composability, and can operate over non-relational data. The monoid representation allows for high-level optimizations, influenced by data mining techniques, that avoid the computation of cross products during data cleaning. The next two sections will present representations that are more suitable for additional optimization tasks.

5. UNIFIED ALGEBRAIC OPTIMIZATION

The result of the optimizations at the monoid comprehension abstraction level is a rewritten comprehension. While the comprehension has undergone optimizations such as filter pushdown and partial unnesting, there are still opportunities for optimizing the overall cleaning task. Therefore, the second abstraction level translates a comprehension into a *nested relational algebra* expression [18], which is more suitable for the next round of CleanM optimizations.

The full algorithm for rewriting a comprehension to an algebraic plan is presented in [18]; the result is a logical plan that uses the operators of Table 1. *Select* and *join* resemble the relational algebra synonymous operators. The *unnest* operators explicitly handle nested data values. The *reduce* and *nest* operators overload the relational projection and grouping operators; they are also responsible for reasoning in terms of different monoid types, and transforming inputs of a specific monoid type (e.g., the k-means monoid) to output of a potentially different monoid type (e.g., a bag / multiset).

There are three major benefits from the algebraic representation: First, there exist rules, which remove any leftover query nestings [18]. Query unnesting is useful in data cleaning, since query and data nestings are inherent in cleaning operations. Second, by expressing all different monoid types into a common, confined algebra, it becomes possible to detect opportunities for intra-operator and inter-operator optimizations, such as work sharing between operators. The running example depicted in Figure 1 shows the first two benefits. Finally, by translating comprehensions into an algebraic form, the optimization techniques that have been proposed in the context of the established relational algebra become applicable over an unnested, simplified query representation.

Optimizations at the algebra level

CleanM queries benefit from many expression simplifications that are possible at query rewrite time [18]. After having removed the nestings of the query, apart from the relational algebra optimizations, the optimizer can detect common patterns and enable work sharing between operators. In the following we present the simplifications that the query of our running example goes through.

The query checks for invalid terms, duplicates, and functional dependency violations. A baseline approach would treat each cleaning operation as a separate task which traverses the input and detects violations. Treating each operation on its own results in the plans A, B, C of Figure 1. Plan A performs term validation via token filtering: It unnests the list of names in order to compute the tokens of each name, then groups by token to detect similar names. By injecting explicit unnest operators, CleanM avoids having to access repeating BLOB-like tuples of the form $(token_i, \{names\})$

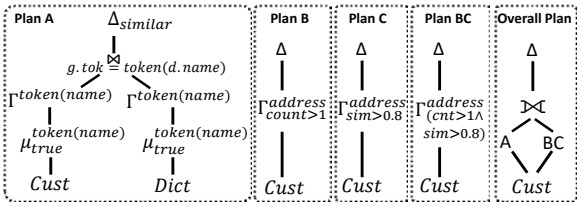

Figure 1: Algebraic plans for our running example, and optimized rewritten plans that coalesce operators and share work.

Operator	Spark Equivalent
σ_p	filter
Δ_p^e	map → filter
μ_p^{path}	flatmap(x→path.filter(y → p(x, y)).map(y→(x,y)))
$\not\mu_p^{path}$	flatmap(x→r=path.filter(y → p(x, y)), if(r.empty) (x, null) else r.map(y→(x,y)))
$\Gamma_p^{\oplus/e/f}$	**aggregateByKey** → mapPartitions
$\bowtie_{f(A)=g(B)}$	join
$\bowtie_{f(A)\ \theta\ g(B)}$	**theta join** → filter
$\not\bowtie_{f(A)=g(B)}$	left outer join
$\not\bowtie_{f(A)\ \theta\ g(B)}$	**theta join** → map

Table 2: Translation of algebraic operators to Spark operators. Bold parts introduce new Spark operators or deviate from the translation that Spark SQL would have performed.

for each element of a nested collection to be processed; it operates over smaller $(token_i, name_j)$ tuples instead [26]. Plan B checks the functional dependency: it computes groups of *address*, and outputs the groups containing more than one phone prefix. Plan C checks for duplicates by again building groups of *address* and checking within each group for entities that are more than 80% similar.

The algebraic rewriter of CleanM detects the commonalities of Plan B and C, and instead produces Plan BC, which coalesces the two grouping passes into one, and applies both filters at once. In addition, given that all the sub-plans scan the same table, the algebraic rewriter produces a DAG-like overall plan, which scans the dataset once, performs the cleaning operations in parallel, and then joins the violating entries of each side using an outer join. In summary, translating cleaning operations into a unifying algebraic form enables, among others, powerful forms of query and data unnesting, coalescing operators, and reducing duplicate work.

6. EXECUTING DATA CLEANING TASKS

The result of optimizations at the algebraic abstraction level of CleanM is a succinct logical plan. The last step of the rewriting process generates a physical plan that is compatible with the execution engine that will perform the data cleaning tasks. This work uses Spark [48] as the scale-out execution substrate, therefore the algebraic plan gets translated to the operators of the Spark API.

Why not Spark SQL? Given that Spark is the current execution engine for CleanM queries, an alternative approach would be to directly map CleanM to the Spark SQL module of Spark [6], which exposes declarative query capabilities and introduces Catalyst, an optimizer over Spark. The Catalyst optimizer, however, assumes tabular data and only considers relational rewrites; it is thus unable to reason about and perform the optimizations suggested so far by this work. Also, the physical Spark plans that Catalyst generates are agnostic to characteristics of real-world data cleaning tasks, namely the facts that i) there is significant skew in the data touched, and that ii) the tasks executed typically require the computation of expensive theta joins. On the contrary, in the final, third abstraction

level, CleanM queries get translated into a physical execution plan which both considers data skew and explicitly handles theta joins.

From nested algebra to Spark operators. Table 2 lists the mapping from the nested relational algebra to Spark operators. The mapping of the *selection* and *reduce* operators is straightforward. The *unnest* operators iterate through a dataset's elements and through a specific nested field of each element. The *Nest* operator, which resembles a SQL *Group By*, is translated into a combination of operators: First, *aggregateByKey* groups data records based on a key. Then, *mapPartitions* applies a function over each partition. Nest optionally evaluates a binary predicate (an equivalent functionality to SQL `HAVING`). In this case, a filter operation also takes place per partition. Finally, the Join operator gets translated into the respective Spark equi-join operator. The handling of other types of joins is more nuanced: By default, Spark SQL and Spark resort to a cartesian product followed by a filtering operation. Given the high frequency of theta joins in the domain of data cleaning, we instead implement an alternative, statistics-aware theta join [36].

Optimizations at the physical level

When translating nested relational algebra operators into a Spark plan, we explicitly consider the presence of i) skew in the data, and ii) theta joins as part of the cleaning process.

Handling data skew. Value distribution in real-world data is rarely uniform. In addition, certain data values can be more susceptible to errors. A cleaning solution must therefore remain unaffected by data skew. In the context of scale-out processing, skew handling is reflected by how one shuffles data in the context of operations such as aggregations. Spark SQL performs sort-based aggregation: it sorts the dataset based on a grouping key, different data ranges of which end up in different data nodes. Then, Spark SQL performs any subsequent computations locally on each node. When, however, some values occur more frequently, the partitions created are imbalanced. Thus, the overloaded nodes lag behind and delay the overall execution. On the contrary, as Table 2 shows, CleanM uses the *aggregateByKey* Spark operator which performs the aggregate locally within each node and then merges the partial results. Thus, CleanM i) minimizes cross-node traffic by forwarding already grouped values, and ii) is more resilient to skew since popular values have already been partially grouped together.

Handling theta joins. In the general case of a join with an inequality predicate, Spark SQL generates a plan involving a cartesian product followed by a filter condition. The result is suboptimal performance when executing theta joins – one of the most frequent operators in data cleaning. We thus implement a custom theta join operator based on the approach of [36]. The new operator represents the cartesian product as a matrix, which it partitions into N uniform partitions. First, the operator computes statistics about the cardinality of the two inputs, which it then uses to populate value histograms. Then, assuming the presence of N nodes, the operator consults the observed value distributions to partition the matrix into N equi-sized rectangles, and assigns each partition to a Spark node. As a result, the operator ensures load balancing; each node checks separately the condition on the partition for which it is responsible.

7. CleanDB: A DATA CLEANING SYSTEM

We validate the three-level design of CleanM by implementing CleanDB, a unified cleaning and querying engine. We build CleanDB by extending RAW [27], an adaptive query engine that operates over raw data. Specifically, we extend the commercial version of RAW [2], which operates over the Spark runtime. CleanDB serves as a replacement layer of Spark SQL [6]; it exposes the expressive power of CleanM without the compromises that

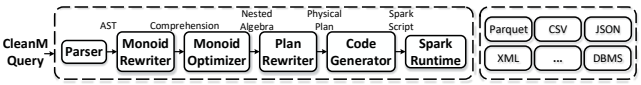

Figure 2: The architecture of CleanDB.

Spark SQL makes. CleanDB optimizes the cleaning operations in a unified way and executes them in a scale-out fashion; the final physical plan is equivalent to handwritten Spark code. The end result is a system that can both query and clean input data. In the following, we present the components of CleanDB which extend the respective components of RAW.

The architecture of CleanDB. Figure 2 presents the components of CleanDB. When receiving a query, the *CleanM parser* rewrites it into an abstract syntax tree (AST). Then, the *Monoid Rewriter* "de-sugarizes" the AST into a monoid comprehension, also considering the monoids presented in Section 4. The *Monoid Optimizer* first applies rewrites over the input comprehension in order to simplify it, push down any filtering expressions, flatten nested comprehensions, unnest existential quantifications, etc. Then, the optimizer rewrites the comprehension into a nested relational algebra, and performs additional rewrites and optimizations over it, such as coalescing multiple operators into a single one.

The output of the Optimizer is a nested relational algebra expression, which the *Physical Plan Rewriter* translates to a plan of physical operators. We plan to extend this level with more low-level "building blocks". Finally, the *Code Generator* dynamically generates the Spark script that represents the input query to reduce the interpretation overhead that hurts the performance of pipelined query engines [33]. After the generation of the Spark script, the Spark Executor deploys the final script in scale-out fashion.

Interestingly, Spark by default associates the result of the execution with the DAG of operations that produced it. We aim to use this built-in data lineage support to incorporate additional data cleaning functionality in future work [20].

8. EXPERIMENTAL EVALUATION

The experiments examine how CleanDB performs compared to the state of the art, while demonstrating the benefits stemming from the three optimization levels of CleanM.

Experimental Setup. We compare CleanDB against BigDansing[1] [29] because it is, to our knowledge, the only currently available scale-out system that explicitly targets data cleaning[2]. We also compare CleanDB against an implementation on top of Spark SQL. Spark SQL uses a relational optimizer to produce query plans, whereas CleanDB uses a monoid-aware, three-level optimizer; we can thus gauge the quality of the CleanM rewrites.

All experiments run on a cluster of 10 nodes equipped with 2 $\times$ Intel Xeon X5660 CPU (6 cores per socket @ 2.80GHz), 64KB of L1 cache and 256KB of L2 cache per core, 12MB of L3 cache shared, and 48GB of RAM. On top of the cluster runs Spark 1.6.0 – the latest version for which BigDansing is intended. Spark launches 10 workers, each using 4 cores and 40GB of memory.

The workload we use involves i) denial constraint checks, ii) duplicate elimination, iii) term validation, and iv) syntactic transformations. Denial constraints are a concept directly related to database design, thus we evaluate them over the TPC-H dataset. We use TPC-H for syntactic transformations as well. We use scale factors 15, 30, 45, 60, and 70 of the *lineitem* table. Each of the five versions comprises 90M, 180M, 270M, 360M, and 420M records

[1] We thank the authors of [29] for giving us access to a binary executable.

[2] SampleClean [46] only operates over query-specific samples.

Type	Parameter(s)	Precision	Recall	F-score
tf	q = 2	100%	97%	98.5%
tf	q = 3	100%	96.8%	98.3%
tf	q = 4	99.9%	95.9%	97.9%
K-means	k = 5	99.9%	95.7%	97.8%
K-means	k = 10	99.9%	94.8%	97.3%
K-means	k = 20	99.9%	94%	96.9%

Table 3: Accuracy of term validation approaches over the DBLP dataset.

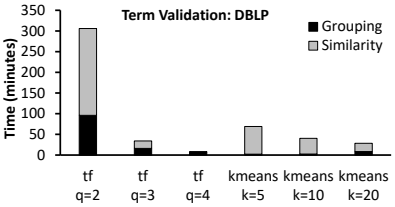

Figure 3: Different configurations of term validation.

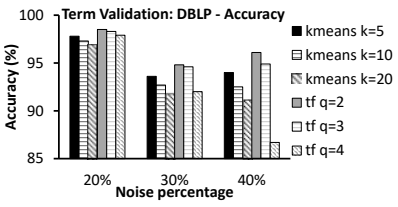

Figure 4: Accuracy of term validation as the noise increases.

respectively. The final dataset size is 11GB, 22GB, 34GB, 45GB, and 52GB respectively. We shuffle the order of the tuples and produce two different datasets by adding noise to 10% of the entries of the *orderkey* and *discount* column respectively. We pick the tuples to edit from the domain of the SF15 version, so that we increase the skew as we increase the dataset size.

We perform duplicate elimination and term validation over the DBLP bibliography hierarchical dataset, because these error categories occur frequently in semi-structured data. We use a subset of DBLP that contains information about articles; each entity contains at most 13 attributes. We add noise to 10% of the author names by a factor of 20%, and scale up the dataset by adding extra entities; we construct new publications by permuting the words of existing titles and by adding authors from the active domain. The end result is a 1GB, a 5GB, and a 10GB XML dataset. We also use the *customer* table of TPC-H because the implementation of duplicate elimination in BigDansing is a UDF that is specific to *customer*. We add duplicate records for 10% of customer entries, where the number of duplicates for each record is a random value generated using Zipf's distribution; the number of duplicates belongs to the intervals [1-50] and [1-100]; respectively. We create the duplicate records by randomly editing the name and phone values. The size of the datasets is 2.2GB and 3.1GB; respectively. We also use the Microsoft Academic Graph (MAG) [41], which is a database of scientific publications stemming from all research areas. We evaluate duplicate elimination over the original version of MAG, since its main issue is the existence of duplicate publications; the same publication may appear multiple times, with variations in the title and DOI fields, or with missing fields. We build MAG by joining the *Paper, Author* and *PaperAuthorAffiliation* datasets. The resulting dataset contains 7 columns and has size 33GB.

We use response time and accuracy (when applicable) as metrics. Response time includes the time taken to read the input, perform a cleaning task, and store the detected violations. In the case of term validation, the output includes both detected violations and suggested repairs. We measure accuracy by verifying the correctness of the repairs against a sanitized version of the dataset.

The rest of this section uses the aforementioned cleaning tasks to visit the CleanM optimization levels, and examines how each of them contributes to the fast and accurate responses of CleanDB.

8.1 Optimizations at the monoid level

CleanDB is the only scale-out data cleaning system that supports term validation; Spark SQL computes the cross product of the input dataset and a dictionary, using a UDF to compute the similarity of each (record, dictionary value) pair, and prune non-similar entries. The overall Spark SQL script was non-interactive in our experiments. This section demonstrates the benefits of the monoid-level optimizations and the importance of calibrating data cleaning tasks in the context of term validation; we examine clustering and filtering operations, and show the effect of calibrating each operation based on dataset characteristics.

Term Validation

Term validation is a challenging operation for CleanDB, because it is very resource-intensive. The next experiment measures the response time and the accuracy of the CleanDB term validation using different filtering algorithms and different parameters for them.

The experiment validates the author names of the flat Parquet version of *DBLP* that contains 6.4M entities using the Levenshtein distance metric. The dictionary that CleanDB consults in order to repair the author names comprises 200K names. The experiment launches different k-means configurations by changing the *number of centers (k)* which it obtains from the dictionary. The same experiment also launches different token filtering configurations using a different *token length parameter (q)*.

Runtime. Figure 3 presents the time taken to clean the author names using k-means and token filtering as pruning methods, while also using different parameters for each method. Each bar comprises the time taken to filter/block the data, and the time to perform the similarity check within the groups. In the case of k-means, using more centers leads to fewer elements in each cluster. Thus, the number of similarity checks decreases. In the case of token filtering, as q increases, performance improves because the tokenization phase produces fewer groups with fewer elements in each one, and thus the number of checks decreases. The token filtering configurations are faster than the k-means ones, except when q=2; the token size proves to be too small and results in too many groups.

Regarding the pre-filtering step, since the tokenization process is expensive, grouping by center is more lightweight than grouping by token. However, the average length of author names in DBLP is 12.8, which is short enough for the tokenization to proceed without significant overhead. Regarding similarity checks, token filtering produces a larger number of smaller-sized groups compared to k-means, thus the total number of pairwise comparisons is smaller. K-means is more sensitive to the statically specified centers.

Accuracy. Table 3 measures the accuracy of the suggested repairs for the term validation task examined. The experiment considers `precision` (i.e., correct updates/total updates suggested), `recall` (i.e., correct updates/total errors) and `F-score` as metrics.

The token filtering configurations are more accurate, because they check the similarity of two author names whenever they have at least one common token. Thus, even if a name is dirty, it will contain at least one clean token that will match a token of the correct name in the dictionary. Increasing q does not hurt accuracy noticeably. K-means becomes less accurate as the number of clusters increases, because similar words end up in different clusters and therefore are not checked for similarity. Still, all the term validation variations of CleanDB exhibit high accuracy.

Figure 4 examines the accuracy of term validation as we vary the noise on the name attribute from 20% to 40%. To obtain a fair comparison, we lower the similarity threshold as we increase the noise, so that we isolate the accuracy of the pruning algorithm and avoid missing results that fail to pass the similarity threshold. The results

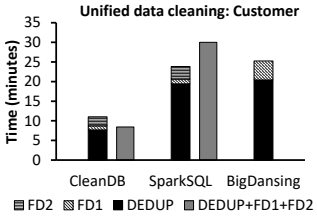

Figure 5: CleanDB rewrites three cleaning operations into a single one, and avoids duplicate work.

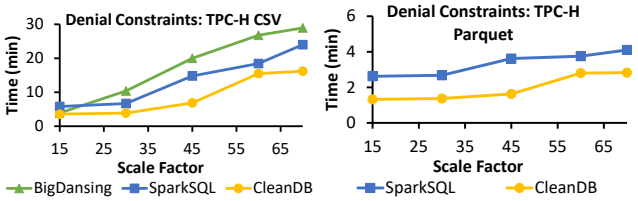

Figure 6: Cost of checking for violations of functional dependencies over TPC-H.

show that accuracy drops slightly as we add more noise. The drop stems from both having lower precision and lower recall. Precision drops because some incorrect matches now pass the low similarity threshold; recall drops because by increasing the noise, two similar words are more likely to get assigned to different groups. However, the drop in accuracy is negligible in all cases but the ones where we have a bigger parameter set for token length q=4 or number of centers k=20; these configurations are more prone to inaccuracies because they produce clusters with fewer items.

Summary. CleanDB can use token filtering and clustering monoids to reduce term validation checks. Both methods avoid false positives, and thus the resulting precision is close to 100%. Calibrating the algorithm parameters enables trading performance for accuracy; still, the accuracy remains above 90% in most cases.

8.2 Optimizations at the algebra level

This section demonstrates the benefits of the algebraic optimizations that CleanDB performs. We focus on how CleanDB optimizes different cleaning operations as a single task.

Unified data cleaning

This experiment resembles our rolling example, and measures the cost of detecting duplicates and functional dependency violations through a single query on the customer dataset; we replace the term validation part of the example with an extra functional dependency, because CleanDB is the only scale-out system supporting term validation. The query in question examines the rules $FD1: address \rightarrow prefix(phone)$, $FD2: address \rightarrow nationkey$ and also checks for duplicate customers given that they appear with the same address. We run the query as i) separate sub-queries and ii) as a single task that also combines the partial results. Figure 5 presents the results.

Results. CleanDB detects that the tasks share a grouping on the *address* field, and thus performs all operations using a single aggregation step. Unifying the cleaning tasks reduces the execution time for CleanDB. BigDansing can only apply one operation at a time, and lacks support for values not belonging to the original attributes (i.e., the result of *prefix()* in *FD1*). Spark SQL is unable to detect the opportunity to group the tasks into one. It starts the cleaning tasks in parallel since they share a common data scan, but then performs a full outer join to combine the output of each operation; unified execution ends up being more expensive than the standalone one. Still, even considering the separate execution, CleanDB outperforms the other systems because of its explicit skew handling when performing FD checks and deduplication.

Transformations

This experiment measures the cost of applying syntactic transformations over the SF70 Parquet version of TPC-H. The experiment examines the added cost when performing lightweight cleaning tasks compared to a traversal of the dataset that projects all its attributes. We consider filling missing values and splitting dates. We fill empty values of the *quantity* attribute using the average value of the existing quantities. We split the *receipt_date* into *day*, *month*, and *year* fields. We also measure the cost of applying the aforementioned operations using a single CleanM query.

Operation	Slowdown
Split date	1.15×
Fill values	1.15×
Split date & Fill values (two steps)	2.3×
Split date & Fill values (one step)	1.19×

Table 4: Overhead introduced by performing syntactic transformations in a plain query. The optimizer of CleanDB applies both operations in one go and reduces overhead by $\sim 2\times$.

Results. Table 4 shows the slowdown that each cleaning task incurs compared to executing the plain query. The individual costs of splitting the dates and filling missing values are almost masked by the query cost. When applying each cleaning operation one after the other, the overall slowdown is computed by adding the overall running times for each dataset traversal. However, CleanDB is able to apply both cleaning operations in one go: The overall cost is then similar to the cost of only applying a single operation, because the execution plan computes the average quantity and then performs both the replacement of missing values and the splitting of the receipt column in a single dataset pass. In summary, CleanDB can intertwine analytics and lightweight cleaning operations, while relying on its optimizer to identify and prune duplicate work.

Summary. Instead of treating each type of cleaning operation as a standalone, black-box implementation, CleanDB optimizes a data cleaning workflow as a whole, identifying optimization opportunities even across different operations. CleanM enables such optimizations because it uses a single abstraction to express all cleaning tasks, and an optimizable algebra as its backend.

8.3 Optimizations at the physical level

This section shows how the physical-level optimizations of CleanDB that focus on handling skew and non-equality predicates accelerate denial constraint and duplicate elimination tasks.

Denial Constraints

This experiment measures the cost of validating two rules. Rule ϕ is a functional dependency which states that the order information of an item determines its supplier. Rule ψ is a general denial constraint stating that an item cannot have a bigger discount than a more expensive item; the filter on price has a selectivity of 0.01%.

$\phi: orderkey, linenumber \rightarrow suppkey$ and

$\psi: \forall t_1, t_2\ t_1.price < t_2.price\ \ \&\ t_1.discount > t_2.discount\ \ \&\ t_1.price < [X]$

The straightforward way to detect functional dependency violations using (Spark) SQL is a self-join query. However, traversing a dataset twice hurts performance. Thus, we benchmark rule ϕ in Spark SQL using a query which groups the data in a way similar

to CleanM. In order to collect all the distinct *l_suppkey* values of each group, we implement a user-defined aggregate function that behaves similar to `GROUP_CONCAT`.

FD Results. Figures 6(a), 6(b) present the time taken to detect violations of ϕ as we increase the size of TPC-H. We present the results for both CSV (Figure 6(a)) and Parquet (Figure 6(b)). Parquet is only supported by CleanDB and Spark SQL; we omit BigDansing in Figure 6(b). The response times of Figure 6(b) are shorter than those of Figure 6(a) because Parquet is a binary columnar optimized data format which also supports compression.

CleanDB is faster than BigDansing and Spark SQL regardless of the underlying format. BigDansing performs hash-based aggregation: it shuffles the data based on a hash function to create blocks that share the same *orderkey* and *linenumber*, and then iterates through each block to check for violations. Spark SQL performs sort-based aggregation: it sorts the entire dataset based on the (*orderkey*, *linenumber*) pair, and different data ranges end up in different data nodes. Then, it performs the aggregate computations locally on each node. Spark SQL outperforms BigDansing because the sort-based shuffle implementation of Spark is more efficient than the hash-based one [47]: The hash-based approach stresses the overall system memory and causes a lot of random I/O, whereas the sort-based approach uses external sorting to alleviate these issues. CleanDB considers data skew when creating the physical query plan: It performs the aggregate operation locally within each data node and then merges the partial results, thus minimizing cross-node traffic. Therefore, CleanDB outperforms the other systems because it translates the query into a set of Spark operators that do not require data exchange until the final merge phase.

Scale Factor	15	30	45	60	70
Time (min)	1.7	2	3.7	4.9	5.65

Table 5: Denial constraints involving inequalities as the dataset size increases. All systems beside CleanDB fail to terminate.

DC Results. The detection of violations of rule ψ involves a self-join that checks the inequality conditions. Table 5 shows that only CleanDB was able to successfully complete the data constraint check. Spark SQL was unable to compute the expensive cross product to evaluate the conditions. BigDansing and CleanDB rely on a custom theta join operator each. The theta join implementation of BigDansing attempts to prune the pairwise comparisons involved in the computation of an inequality join by first partitioning the data, then computing min-max values per partition, and then only cross-comparing partitions whose min-max ranges overlap. The number of avoidable checks, however, is not guaranteed to be high, unless the partitioning of the first step can be fully aligned with the fields involved in the DC; indeed, excessive data shuffling makes BigDansing non-responsive for rule ψ. On the contrary, CleanDB spends more effort to obtain global data statistics, and does a better job balancing the theta join load among the Spark executors.

Duplicate Elimination

The following experiments evaluate duplicate detection over DBLP, MAG and TPC-H customer table; the duplicate detection implementation of BigDansing is specific to the customer table.

We demonstrate the importance of being able to handle heterogeneous datasets by considering multiple different representations for DBLP: We consider i) a JSON version, which has become the most popular data exchange format, ii) a Parquet version that preserves data nestings, iii) a "flat" CSV version, and iv) a "flat" Parquet version. We obtain the last two versions by flattening the entities of the nested input, that is, if a publication has more than one author, then

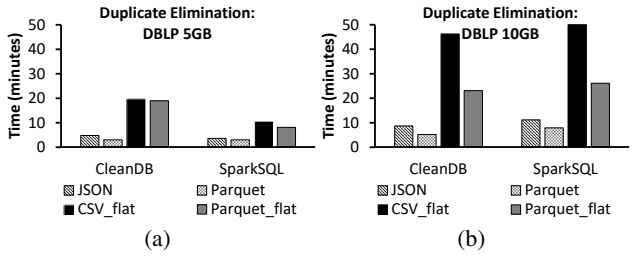

Figure 7: Duplicate elimination over different representations of DBLP. We simplified the dataset for Spark SQL to terminate.

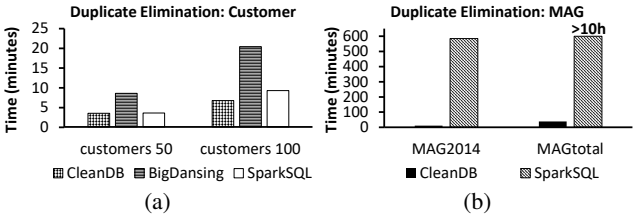

Figure 8: Duplicate elimination over Customer and MAG.

the publication appears in multiple records – one for each author; a common practice followed by relational systems. We compare the response time of CleanDB against Spark SQL. We consider two DBLP publications to be duplicates if they appear on the same journal, have the same title, and the similarity of their attributes exceeds 80% – we assume that the title and journal attributes are "cleaner" than the rest. Both CleanDB and Spark SQL create blocks based on the journal and title values to reduce pairwise comparisons. Similarly, two MAG publications are duplicates if they appear on the same year, have the same author id, and are more than 80% similar.

DBLP Deduplication Results. Spark SQL initially was unable to complete the elimination task, even for an input size of 1GB, because it is sensitive to data skew. Therefore, we removed the most frequently occurring titles from the dataset to obtain a more uniform version and enable the comparison against Spark SQL. The size of the uniform dataset varies from 5GB to 10GB when stored as XML, and the number of entries ranges from 6.4 to 64 million. For the JSON, nested Parquet, "flat" CSV, and "flat" Parquet versions, the size reached 7GB, 2GB, 14GB, and 2.4GB respectively.

Figure 7 presents the response time of the systems that are able to process DBLP. Both CleanDB and Spark SQL are faster when running over the nested JSON and Parquet representations, because flattening the data introduced many more tuples to be processed; thus, being able to operate over the original, non-relational data representation can be a significant asset for many use cases.

Regardless of format, Spark SQL exhibits lower response times for the 5GB case, yet scales less gracefully and is slower than CleanDB for the 10GB version. The explanation for this behavior resembles the one for DCs: Spark SQL uses sort-based shuffling based on the *journal, title* attributes to assign the records of each group into the same partition and then computes the similarity within each group. On the contrary, CleanDB aggregates data locally, and then merges the partial results together. The physical rewrites of CleanDB reduce network traffic and are resilient to skew. However, in the simplified dataset versions that we produced to be able to use Spark SQL, data ends up following a uniform distribution, thus favoring Spark SQL. Still, when the data size increases, some of the values again occur more frequently than others; Spark SQL creates imbalanced partitions which overload some

nodes and thus delay the overall execution time because they have to perform more similarity checks than other nodes.

Customer Deduplication Results. Figure 8(a) presents the response time of all systems over the customer dataset. BigDansing and Spark SQL perform poorly because of the suboptimal way in which they construct the value blocks to be checked for duplicates; instead of grouping values locally and then shuffling them to other nodes, they shuffle the entire dataset. CleanDB scales better than the other systems because of its explicit skew handling.

MAG Deduplication Results. Figure 8(b) presents the response time of all systems over the MAG dataset. Spark SQL was unable to execute the task for the whole dataset, thus we also consider a 6.3GB subset which contains publications from year 2014. MAG is a real-world, highly skewed dataset; CleanDB uses skew-resilient primitives, and thus significantly outperforms Spark SQL.

Summary. The physical-level optimizations, namely support for data skew and theta joins, ensure that CleanDB scales gracefully, and handles realistic datasets for which its competitors are unable to terminate successfully. The experiments also show the importance of allowing data cleaning over the original, intended data format; cleaning nested data proved to be faster when considering the original nested representation instead of flattening all entries.

9. CONCLUSION

Practitioners typically perform manual data cleaning or resort to a number of different cleaning tools – one per error type. Being forced to use multiple tools is inconvenient, makes it hard to apply data cleaning operations iteratively until the user considers data quality to be satisfactory, and seldom guarantees that a cleaning script will be efficiently optimized and executed as a whole.

This work introduces CleanM, a declarative query language that allows users to express their different cleaning scripts. CleanM exposes a wide variety of parameterizable data cleaning primitives which a user can apply over her data. CleanM relies on a powerful, parallelizable query calculus, and a three-level optimization process; all the operations included in a cleaning script are translated to the calculus, and then optimized as one unified task.

We implement CleanDB, a scale-out querying and cleaning framework. CleanDB exposes the functionality of CleanM over multiple types of data sources. CleanDB scales better than existing data cleaning solutions, and handles cases that other systems lack support for / are unable to serve due to performance issues.

Acknowledgments. We would like to thank the reviewers for their valuable comments and suggestions on how to improve the paper. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa).

10. REFERENCES

[1] KNIME. https://www.knime.org/.
[2] RAW Labs. http://www.raw-labs.com/.
[3] Z. Abedjan et al. DataXFormer: A robust transformation discovery system. In *ICDE*, 2016.
[4] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 241–252, New York, NY, USA, 2012. ACM.
[5] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB*, 9(3), 2015.
[6] M. Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.
[7] L. Berti-Équille, T. Dasu, and D. Srivastava. Discovery of complex glitch patterns: A novel approach to quantitative data cleaning. In *ICDE*, 2011.
[8] L. Berti-Équille, J. M. Loh, and T. Dasu. A masking index for quantifying hidden glitches. *Knowledge and Information Systems*, 44(2):253–277, Aug. 2015.
[9] J. Bleiholder and F. Naumann. Declarative data fusion – syntax, semantics, and implementation. In *ADBIS*, 2005.
[10] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *PVLDB*, 7(13), 2014.
[11] X. Chu et al. KATARA: Reliable Data Cleaning with Knowledge Bases and Crowdsourcing. *PVLDB*, 8(12):1952–1955, 2015.
[12] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.
[13] J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.
[14] M. Dallachiesa et al. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, 2013.
[15] T. Dasu and J. M. Loh. Statistical distortion: Consequences of data cleaning. *CoRR*, abs/1208.1932, 2012.
[16] W. Fan. Data quality: From theory to practice. *SIGMOD Record*, 44(3):7–18, Dec. 2015.
[17] L. Fegaras. Incremental query processing on big data streams. *TKDE*, 28(11):2998–3012, 2016.
[18] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *TODS*, 25(4):457–516, Dec. 2000.
[19] H. Galhardas. Data Cleaning and Transformation Using the AJAX Framework. In *GTTSE*, 2005.
[20] H. Galhardas et al. Improving data cleaning quality using a data lineage facility. In *DMDW*, 2001.
[21] F. Geerts et al. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.
[22] D. Haas et al. Wisteria: Nurturing scalable data cleaning infrastructure. *VLDB*, 8(12):2004–2007, 2015.
[23] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.
[24] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String Similarity Joins: An Experimental Evaluation. *PVLDB*, 7(8):625–636, 2014.
[25] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *SIGCHI*, 2011.
[26] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12), 2016.
[27] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014.
[28] M. Karpathiotakis et al. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*, 2015.
[29] Z. Khayyat et al. BigDansing: A System for Big Data Cleansing. In *SIGMOD*, 2015.
[30] W. Kim. On Optimizing an SQL-like Nested Query. *TODS*, 7(3):443–469, 1982.
[31] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.
[32] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.
[33] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.
[34] S. Lohr. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights, The New York Times, 2014.
[35] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. KDD, 2000.
[36] A. Okcan and M. Riedewald. Processing Theta-joins Using MapReduce. In *SIGMOD*, 2011.
[37] L. Passing et al. Sql-and operator-centric data analytics in relational main-memory databases. *EDBT*, 2017.
[38] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *PVLDB*, pages 381–390, 2001.
[39] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB*, 7(12):1059–1070, 2014.
[40] K.-U. Sattler, S. Conrad, and G. Saake. Adding conflict resolution features to a query language for database federations. *AJIS*, 8(1), 2000.
[41] A. Sinha et al. An Overview of Microsoft Academic Service (MAS) and Applications. WWW '15 Companion, New York, NY, USA, 2015. ACM.
[42] W. M. Soon, H. T. Ng, and C. Y. Lim. A Machine Learning Approach to Coreference Resolution of Noun Phrases. *Computational Linguistics*, 27(4):521–544, 2001.
[43] M. Stonebraker et al. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.
[44] R. Verborgh and M. D. Wilde. *Using OpenRefine*. Packt Publishing, 2013.
[45] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, Mar. 1985.
[46] J. Wang et al. A Sample-and-clean Framework for Fast and Accurate Query Processing on Dirty Data. In *SIGMOD*, 2014.
[47] R. Xin. Made sort-based shuffle the default implementation, Spark Issue 3280, 2014.
[48] M. Zaharia et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.