Thèse n° 7210

Holistic, Efficient, and Real-time Cleaning of Heterogeneous Data

Présentée le 19 novembre 2021

Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Styliani Asimina GIANNAKOPOULOU

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. A. Ailamaki, directrice de thèse
Prof. I. F. Ilyas, rapporteur
Dr D. Srivastava, rapporteur
Prof. K. Aberer, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

Inspiration exists, but it must find you working.
— Pablo Picasso

To my parents Ioannis and Vassiliki, and my sister Panagiota

Acknowledgements

During my PhD journey I was fortunate enough to be surrounded by people who supported me and helped in making this thesis possible.

First and foremost, I would like to thank my advisor, Anastasia Ailamaki. Anastasia is an inspiring person who is really enthusiastic about research, and she has the unique ability to transmit her excitement to the people around her. Her students are her top priority; one of her main characteristics is that she encourages and motivates her students to exit their comfort zone and always aim for excellence in research. Anastasia was always there supporting me, even in the most difficult times of my PhD. Her advice, both in research, as well as in life was invaluable. I have learned a lot while being her student, and I really appreciate the opportunity she gave me to be a member of the DIAS family.

I would also like to thank the jury members of my thesis for investing the time to read my thesis and provide detailed feedback. Specifically, I would like to thank Ihab Ilyas whose work has highly influenced the first two parts of my thesis, Divesh Srivastava and Karl Aberer whose detailed feedback helped in shaping the thesis. I also thank Aikaterini Argyraki for agreeing to be the president of my thesis committee. I am also very grateful to Surajit Chaudhuri and Vivek Narasayya for being my mentors during my internship in Microsoft Research; our collaboration and our discussions during my internship have significantly broadened my knowledge and experience in the field.

During my PhD I was immensely lucky to be a member of the amazing DIAS family. I would like to thank Danica, Renata, Iraklis, Manos, Matt, Eleni, Mira, Giorgos, Utku, Periklis, Panos, Aunn, Viktor, Eleni, Raja, Darius, Satya, Tahir, Odysseas, Angelos, Foteini, Giagkos, Bikash, Srinivas, and Haoqiong for always being available to talk and to give useful feedback. I am especially thankful for having met and collaborated with a set of exceptional people in DIAS: Manos Karpathiotakis is the best advisor one could ever have during their PhD; I am thankful for his guidance, friendship, and for always being available to help and brainstorm with me, he is a great source of knowledge, without him this thesis would not be possible. Angelos Anadiotis is a great researcher with whom we had very fruitful discussions during my last years. I am also very grateful for having collaborated with Giagkos and Eleni from whom I learned a lot. Matt is a great friend, always caring for me, I really enjoyed TAing with him, but I especially loved his late-night singing which would bring extra joy during difficult times. Panos is one of

the most amazing collaborators and friends that gives the best feedback, I was really lucky for having him around. Finally, I'm thankful to Erika Raetz, for the help with all the administrative stuff at EPFL, and Dimitra Tsaoussis thank you for your friendship and for always being there to help me even with last minute requests.

During my stay in Lausanne I had the chance to meet people who became my family away from home. I would like to thank "*les Lausannoises*", that is Aggeliki, Angelina, Sylvia, and Chrystalleni for bringing joy to my days through our awesome discussions and our laughs. I am also thankful for the most amazing friends I made: Manos, Eleni, Matt, Prodromos, Marios, Panos, Dimitris, Pavlos, Nathalie, Alexandros, Kyveli, Tiziano, Kristina, Ivi, Errikos, and Sophia; I really enjoyed all the lovely times we spent together.

Despite living in Lausanne, my Greek friends have never stopped keeping me company both virtually, as well as in-person whenever I would visit Greece. Ioanna is the best friend one can have, always cares for me. Anny and Elpida who would always reach out to me to check how I am doing. Finally, I really enjoyed whenever Eirini and Tasos would visit Lausanne, as well as our nice trips around Europe.

Having Panayiotis by my side throughout all these years was the most important support and encouragement I had. Panayiotis had the ability to propose simple solutions even when I was desperate. Also, I was extremely lucky to have a little comedian in the house who would do anything to make me smile even in the most difficult times. I am really grateful of his company during this journey. Thank you!

Last but not least I would like to thank my lovely family, my father Giannis, my mother Vasso and my sister Gioula for their unconditional love and support throughout my life and especially throughout my PhD life. All three of them are hard-working, stick to their goals and constitute an exemplar for me. The main inspiration for pursuing a PhD came through my father who was always excited about research and unsolved problems. I would also like to thank my sister's husband Stamatis and my little nieces Theodora and Vassiliki for bringing extra joy and happiness to my life.

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa).

Lausanne, July 14, 2021 S. G.

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. Existing solutions that attempt to automate the data cleaning procedure treat data cleaning as a separate offline process that takes place before analysis begins, while also focusing on a specific use-case. In addition, when the analysis involves complex non-relational data such as graphs, data cleaning becomes more challenging as it involves expensive operations. Therefore, offline, specialized cleaning tools exhibit long running times or fail to process large datasets. At the same time, applying data cleaning before analysis starts assumes a priori knowledge of the inconsistencies and the query workload, thereby requiring effort on understanding and cleaning data that is unnecessary for the analysis. Therefore, from a user's perspective, one is forced to use a different, potentially inefficient tool for each category of errors.

In this thesis we aim for *coverage* and *efficiency* of data cleaning. We design and build data cleaning systems that employ high-level abstractions to (a) represent and optimize different cleaning operations for data of various formats, and (b) allow for real-time data cleaning that is relevant to data analysis.

We introduce CleanM, a language that 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.

To further reduce the data-to-insight time, we propose an approach that performs probabilistic repair of denial constraint violations on-demand, driven by the exploratory analysis that users perform. We introduce Daisy, a system that seamlessly integrates data cleaning into the analysis by relaxing query results. Daisy executes analytical query-workloads over dirty data by weaving cleaning operators into the query plan.

To cover complex data such as graphs, we optimize the building block of data cleaning operations on graph data, that is subgraph matching. Subgraph matching constitutes the main bottleneck when cleaning graph data as it is an NP-complete problem. To optimize subgraph matching, we present a scale-up, radix-based algorithm that starts from an arbitrary parti-

tioning of the graph and coordinates parallel pattern matching to eliminate redundant work among the workers. To address load imbalance, we employ a work stealing technique, specific to the subgraph matching problem. Worker threads steal work from straggler threads by using a heuristic that maximizes the work stolen, while at the same time preserves the order of evaluating candidate vertices.

Overall, instead of using offline, specialised tools, this thesis designs abstractions that optimize different cleaning primitives over heterogeneous data, while also integrating data cleaning tasks seamlessly into data analysis. Specifically, we provide (a) a declarative high-level interface backed by an optimizable query calculus and (b) the required optimizations at the underlying data cleaning layers while also taking into consideration the exploratory analysis that users perform.

Keywords data cleaning, denial constraints, duplicate elimination, data transformations, integrity constraints, data analysis, data preprocessing, monoid comprehensions, scale-out cle-aning, cleaning heterogeneous data, graph databases, graph pattern matching

Résumé

Le nettoyage des données est devenu un élément indispensable de l'analyse des données en raison du quantité croissante de données erronées. Les data scientists passent la plupart de leur temps à préparer des données sales avant qu'elles puissent être utilisées pour l'analyse. Solutions existantes qui tentent d'automatiser le nettoyage des données traitent le nettoyage des données comme un processus qui se déroule hors ligne et a lieu avant le début de l'analyse, tandis que se concentrant en même temps sur un cas d'utilisation spécifique. De plus, lorsque l'analyse implique des données non relationnelles complexes telles que des graphiques, le nettoyage des données devient plus difficile car cela implique des opérations coûteuses. Par conséquent, les systèmes de nettoyage qui sont spécialisés et fonctionnent hors ligne présentent de longues durées d'exécution ou ne reussissent pas à traiter de grands ensembles de données. En même temps, pour appliquer le nettoyage des données avant le début de l'analyse, on suppose une connaissance a priori du incohérences et la charge de travail des requêtes, ce qui nécessite des efforts de compréhension et de nettoyage des données inutiles pour l'analyse. Par conséquent, par le point de vue de l'utilisateur, on est obligé d'utiliser un système différent, potentiellement inefficace, pour chaque catégorie d'erreurs.

Dans cette thèse, nous visons sur la *couverture* et l'*efficacité* du nettoyage des données. Nous concevons et mettons en œuvre des systèmes qui utilisent des abstractions de haut niveau pour (a) représenter et optimiser les opérations de nettoyage hétérogènes sur des données hétérogènes, et (b) des données propres en ligne qui sont pertinentes pour l'analyse des données.

Nous introduisons CleanM, un langage qui peut exprimer plusieurs types d'opérations de nettoyage. CleanM passe par un processus de traduction à trois niveaux afin de l'optimisation ; une famille d'optimisations différente est appliquée à chaque niveau d'abstraction. Ainsi, CleanM peut exprimer des tâches de nettoyage de données complexes, les optimiser de manière unifiée et les déployer de manière évolutive.

Pour réduire davantage le data-to-insight temps, nous proposons une approche qui effectue une réparation probabiliste des violations de contraintes de refus qui fonctionne on demand, guidée par l'analyse exploratoire des utilisateurs. Nous introduisons Daisy, un système qui intègre de manière transparente le nettoyage des données dans l'analyse en assouplissant les résultats des requêtes. Daisy exécute de requête analytiques sur des données erronées en intégrant les opérateurs de nettoyage dans le plan de requête.

Pour traiter des données complexes telles que les graphiques, nous optimisons le bloc de construction des opérations de nettoyage des données sur les données de graphes, c'est-à-dire le match de sous-graphes. Le processus de match de sous-graphe est le principal goulot quand on nettoie des données graphiques car il s'agit d'un problème NP-complet. Nous présentons un algorithme scale-up, basé sur le radix, qui part d'un partitionnement arbitraire du graphe et coordonne la correspondance de motifs parallèles à éliminer le travail redondant parmi les travailleurs. Pour remédier au déséquilibre de charge, nous utilisons une technique de vol de travail, spécifique au problème de match de sous-graphes. Les threads de travail volent le travail des threads retardataires en utilisant une heuristique qui maximise le travail volé, tout en préservant l'ordre d'évaluation des sommets candidats.

Dans l'ensemble, au lieu d'utiliser des outils spécialisés hors ligne, cette thèse conçoit des abstractions qui optimisent différents primitives de nettoyages sur des données hétérogènes, tout en intégrant de manière transparente les tâches de nettoyage des données dans l'analyse de ces données. Plus précisément, nous fournissons (a) une interface déclarative de haut niveau soutenue par un calcul de requête optimisable et (b) les optimisations requises au niveau des couches sous-jacentes de nettoyage de données, tout en tenant également compte de l'analyse exploratoire que les utilisateurs effectuent.

Mots clés nettoyage des données, contraintes de refus, élimination des doublons, transformations de données, contraintes d'intégrité, analyse des données, prétraitement des données, compréhensions monoïdes, nettoyage à grande échelle, nettoyage de données hétérogènes, bases de données de graphes, correspondance de modèles de graphes

Contents

List of Figures

List of Tables

1 Introduction

Data science has given a big push to the penetration of computer science into a vast variety of disciplines to extract meaningful insights. Information extraction increasingly relies on data collected in unprecedented volumes and variety. Data scientists start from lakes of raw data, usually blindly collected and stored in batches by automated processes, and they apply different types of analysis tasks, ranging from relational to data mining and machine learning operations. The processes of gathering, storing, and integrating diverse datasets introduces, however, several inaccuracies in the data: Analysts spend 50%-80% of their time preparing dirty data, before it can be used for information extraction [81]. Depending on the accuracy requirements of the workload and the data they need to access, data scientists have to iteratively apply cleaning tasks, until they are satisfied with the resulting quality. Therefore, data cleaning is a major hurdle for data analysis.

1.1 The Need for Clean Data

Data cleaning is a necessary preparatory step because the existence of errors in data induces inaccuracies in data analysis [74]. Dirty values affect the data distribution, as an analysis based on inaccurate aggregates or on training over inaccurate distributions might reveal wrong trends, or even hide an existing trend. Apart from the distribution, an analysis based on machine-learning models is sensitive to the relationships among entities. As a result, erroneous values could cause the machine-learning model to underfit hence show poor performance. Therefore, to extract accurate insights, data scientists need to first fix the errors before running their analyses.

The impact of dirty values are linked to every individual analysis task, as well as the underlying data. Depending on the task, one dirty value might be more important compared to another dirty value, hence it might have a stronger impact on the analysis result. The importance of an error depends on three reasons: First, an analysis task might access only a specific part of the dataset, consequently errors in the unnecessary part of the data can be ignored. Second, data continuously changes, thus, errors in stale or old data do not affect the analysis.

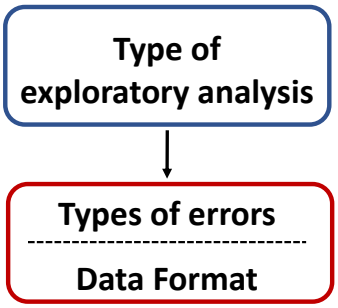


Figure 1.1 – Data cleaning challenges.

Finally, several machine learning algorithms such as In addition to the importance of different errors for the analysis, depending on the end-user's perception of cleanliness, the preparatory work might involve different cleaning operations in order to enable the analysis to take place. Therefore, there is need for real-time data cleaning while exploring the data, by also taking into consideration the variability in the types of errors, as well as in the types of data.

1.2 Data-Cleaning Challenges

The data cleaning process consists of two steps: error detection and data repair. Error detection involves operations such as checking the satisfiability of qualitative rules, identifying patterns and similar entities. Data repair takes place over the detected errors and is the process of correcting the identified errors by replacing dirty values with the correct ones. Both error detection and data repair tasks are challenging; they first require exploratory queries to identify possible errors, and then one needs to use multiple cleaning tools, hand-written scripts, and manual effort to clean the data.

The data cleaning problem involves challenges spanning across three dimensions, as shown in Figure 1.1. Specifically, at the data level, the types of errors that exist, as well as the the data format, affect the detection and repair of errors. Each error type and data format requires their specific representations of the cleaning task.. Apart from the representation, each case needs particular optimizations that are specific to the data type or the operation. Cleaning tasks also need to take into consideration the type of analysis that users perform in order to avoid executing unnecessary pre-processing work.

Variability of Errors. Data cleaning is challenging because errors arise in different forms: Syntactic errors involve violations, such as values out of domain and range. Semantic errors are also frequent in non-curated datasets. They involve violations of semantic data relations, e.g., Beijing is located in the US. Semantic errors are more challenging because they involve inconsistencies in real-life data and might require auxiliary data sources for their detection. Apart from semantic relation violations, the presence of duplicate entries is a typical issue when integrating multiple data sources. The aforementioned data cleaning tasks translate into a broad range of operations that contain inequality joins, similarity joins, or iterative

operations as building blocks. Consequently, it is difficult to build general-purpose systems that can capture the majority of error types and at the same time perform data cleaning in a scalable manner.

Exploratory Analysis. Apart from the existence of various types of errors, data cleaning is an interactive and exploratory process that involves expensive operations. Error detection requires multiple pairwise comparisons to check the satisfiability of the rules [50]. Data repairing adds an extra overhead, as it requires many iterations of assigning candidate values to dirty cells, until all the rules are satisfied [27, 32]. Data scientists also detect inconsistencies and constraints at data exploration time [10]. Hence, traversing the whole dataset multiple times to repair each discovered discrepancy is cost prohibitive.

Variability of Data Format. Data heterogeneity poses another challenge, as the complexity of data formats has a direct, negative, impact on the efficiency of data cleaning tasks. Apart from relational, structured data, one might need to clean semi-structured (e.g., JSON, XML) or even unstructured data (e.g., graphs). Semi-structured and unstructured data are inherently skewed because some popular entities can be connected with multiple other entities. Furthermore, detecting violations of integrity constraints in the case of graph data involves subgraph matching as a building block [41, 42]; to detect inconsistencies there is need to specify a topological constraint first to restrict the subgraphs of interest, and then identify rule violations over the selected subgraphs. Subgraph matching is a known NP-complete problem as it is equivalent to the subgraph isomorphism problem [30]. Dealing with high-complexity operations, such as subgraph matching further impedes the data cleaning process.

1.3 Efficiency and Coverage Issues of Existing Cleaning Approaches

State-of-the-art data cleaning approaches can be classified based on two dimensions: (a) their functionality, and (b) whether they are offline approaches or they are online analysis-aware.

In terms of data cleaning functionality, existing systems can be further divided into two categories. The first category includes interactive tools through which a user specifies constraints for the columns of a dataset or provides example transformations. Such tools include Potter's Wheel [101] and Data Wrangler [62]. 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 that enable several data cleaning operations [32, 48, 66, 103, 116]. Both categories lack a universal representation for users to express various cleaning scripts, and/or are unable to optimize the different cleaning operations as one unified task, because they treat each operation as a black-box UDF.

In addition to functionality, data cleaning depends on the analysis that data scientists perform. State-of-the-art approaches can be divided into offline, and online analysis-aware approaches. Offline tools [32, 66, 103] treat data cleaning as a separate process, decoupled from analysis.

Applying data cleaning before analysis begins requires prior knowledge of the errors that exist. Offline cleaning is also cost-prohibitive, as it operates over the whole dataset [40]. Analysis-aware tools [10, 129, 74, 24] focus on entity resolution or deduplication, or they limit themselves to cell-level errors. Whereas, entity resolution tools either require expensive preprocessing [10] or support only approximate answers for aggregate queries [129].

1.4 Thesis Statement and Contributions

In this thesis, we address the coverage and efficiency of data cleaning, and we redesign the data cleaning stack to optimize it for the workload and the underlying data. Our goal is to enable and optimize different data cleaning operations over data of various formats, while weaving cleaning into the analysis task at hand.

Thesis Statement

Cleaning dirty data is a costly process that depends on the analysis that users perform. Existing data cleaning approaches focus on a single error-category and exhibit long running times or are dependent on a specific data format. Instead of using inefficient, specialised tools, we design abstractions that cover and optimize different cleaning primitives over multiple types of data, while also integrating data cleaning operations seamlessly into data analysis.

1.4.1 The End goal: Functionality and Efficiency

To address the requirements for data and workload heterogeneity when cleaning data online, there is need for a paradigm shift. Therefore, depending on the workload and the underlying data there is need for both a high-level representation for data cleaning, as well as low-level, cleaning-aware optimizations.

The high-level representation serves a purpose similar to that of SQL for data management in terms of expressivity and optimization: First, SQL enables 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 that depends on the specific needs of the data, such as the errors data contain and the data format, hence it requires a language that enables 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.

As data cleaning is an expensive exploratory process, there is need for an efficient cleaning approach that is weaved into the exploratory analysis and that cleans data on-demand. On-the-fly cleaning repairs only necessary data, as a result if only a subset is analyzed, the wasted-effort is minimized. Online cleaning also benefits offline cleaning by enhancing the predictability on the required cleaning tasks. Therefore, integrating cleaning with analysis efficiently supports exploratory applications [33] by reducing data-to-insight time.

To generalize data cleaning over various formats, including unstructured information, a data cleaning approach needs to support graph data. Graph data encapsulate both structured as well as unstructured information. Unlike relational and hierarchical data, cleaning unstructured data results in high complexity because error detection involves pattern matching as a building block [41]. Specifically, checking for inconsistencies requires the discovery of specific subgraphs that should satisfy a given rule, or that should be similar. Thus, before checking the validity of any condition, pattern matching is necessary. Parallelizing pattern matching alleviates the high complexity of the problem but still requires expensive partitioning and synchronization tasks in order to exploit the parallelization opportunity. Therefore, there is need for a parallel pattern-matching approach that (a) synchronizes work by avoiding the execution of duplicate work by the workers, (b) it prunes redundant intermediate results that fail to lead to a qualifying result, and (c) it efficiently balances the pattern-matching load by taking into consideration the presence of skew in graphs.

To this end, we make the following contributions:

An Optimizable Query Language for Unified Error Detection: We introduce 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* [44]. The monoid calculus is an optimizable calculus that is inherently parallelizable and can also represent complex operations between various data collection types. Then, comprehensions are translated into an intermediate algebra that enables opportunities for inter-operator optimizations and for work sharing. Finally, the algebraic operators are translated into a physical plan that 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, JSON, etc.), the overall task will be treated as a single query, optimized as a whole, and executed in a distributed, scale-out fashion.

Real-Time Cleaning of Denial Constraint Violations through Relaxation: We present the first approach that intermingles cleaning denial constraint violations with exploratory Select-Project-Join (SPJ) and aggregate queries, and that gradually cleans the data. Denial constraints (DCs) comprise a family of rules that have been widely used to capture real-life data inconsistencies [108, 40]. To provide correct results over dirty data, we introduce cleaning operators in the query plan and employ a cost model to optimally place them. To enable cleaning operators to detect errors, we define at the execution level a novel query-result relaxation mechanism in the context of denial constraints. Query-result relaxation enhances the query result with correlated data from the dataset to allow error detection. Then, given the detected errors, we propose candidate fixes by providing probabilistic results [117]. We validate our approach by building CleanDB, a distributed incremental cleaning framework over Spark [132].

Scalable Cleaning of Graph Data: We propose a parallel, online graph pattern matching approach that progressively merges partial patterns through work-sharing. To enable information sharing among workers, we use a radix tree, through which workers progressively discover qualifying patterns or partial patterns with no valid continuation, even if they span across multiple partitions. The execution of pattern matching through the radix tree also eliminates redundant work because each worker is responsible for evaluating a different part of the pattern To solve the load-imbalance problem, we apply online work stealing by integrating it in the optimization process. We show that work-stealing fully compensates for balanced, workload-aware graph partitioning without additional cost.

1.4.2 Thesis Roadmap

This thesis is organized as follows:

- In Chapter 2, we provide the required preliminaries for this thesis, as well as related work and how this thesis advances the state-of-the-art.
- In Chapter 3, we present a unified, scale-out approach for data cleaning that represents different error detection tasks and optimizes them as a whole.
- In Chapter 4, we present an approach that efficiently intermingles cleaning operations in exploratory queries by relaxing query results in a way that the cleaning operators clean only the relevant data for the query.
- In Chapter 5, we describe an approach that efficiently scales up pattern matching tasks over graph data.
- In Chapter 6, we summarize the big picture by concluding the thesis and proposing future directions.

2 Related Work: The data cleaning ecosystem

This chapter presents the necessary background knowledge for this thesis, as well as the related state-of-the-art work in the data cleaning domain, and shows how this thesis advances the field.. First, we define the data cleaning operations that are commonly used from data scientists to clean their data. Then, we survey the related work categorized by the type of cleaning they support, by whether they are analysis-aware, and based on the optimizations they perform for graph data.

2.1 Data cleaning operations

In the following we present a set of data cleaning operations that data scientists use to clean their data. Such operations provide different levels of expressiveness in order to allow the detection of common discrepancies that appear in business or academic data [40, 59]. Each type of data cleaning operation is used to discover a different type of discrepancy. Overall, the data cleaning operations can be classified into detecting qualitative violations using constraints, detecting similar items and performing transformations over the data.

Denial Constraints (DC)

The family of *denial constraints* [40] contains universally quantified first order language sentences that represent data dependencies, such as functional dependencies and conditional functional dependencies. DCs have the following form:

$$\forall t_1, ..., t_k \neg (p(x_1) \wedge p(x_2) \wedge ... p(x_m))$$

where each t_i is a tuple, each p_i is a predicate involving conditions between the attributes of one or more tuples, k is the number of involved tuples, and m is the number of predicates. If a dataset contains one or more tuples for which the predicates $p(x_1)...p(x_m)$ hold, it is considered to be inconsistent.

The most common subcategory of denial constraints is functional dependencies which are defined as follows. Given a schema S a functional dependency is of the form $X \rightarrow Y$, where $X \in attr(S)$ and $Y \in attr(S)$ which means that the values of attributes in X determine the values of attributes in Y, that is for any two tuples t_1, t_2 iff $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$.

Duplicate Elimination

Duplicate elimination involves the discovery of tuples that refer to the same real-world entity [72]. 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.

Transformations & Term Validation

Transformations involve applying a formula to a set of values, or mapping values to a set of semantically related values [7]. 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.

2.2 Use-case specific data cleaning approaches

For each error category, there are several approaches that propose error detection and data repairing algorithms that allow to clean the errors efficiently. In the following, we present data cleaning approaches for each type of error that follow either an interactive or a (semi-) automatic approach.

Interactive Data Cleaning

Potter's Wheel [101], OpenRefine [127], and Trifacta – the commercial version of Data Wrangler [62] – are established interactive data cleaning systems. Potter's Wheel [101] provides a spreadsheet-like interface via which the user gradually repairs her dataset. The user thus performs transformations, such as merging two columns, and at the same time, a background daemon process detects potential syntactic errors. For the daemon to detect any errors, a user has to first specify patterns to which values must adhere, a set of domains to which data

entries must belong, and the constraints of each domain. After the user has specified all the required transformations, the system compiles them into an optimized program, which can be applied on the current dataset, or can be invoked on other similarly structured datasets.

Wrangler [62] extends the transformation language of Potter's Wheel to allow users to specify more expressive transformations, such as filling missing values. Wrangler simplifies the transformation process by also suggesting transformations based on the history of user operations. OpenRefine [127] exposes a spreadsheet-like interface as well, via which it operates on multiple data formats and allows transformations on rows and columns. OpenRefine enables transformations through facets that define filtering criteria, e.g., keep a subset of cells of a column that satisfy a constraint. Pentaho [97], Knime [69], and Paxata [96] allow for more complex operations, but rely on black box UDFs.

This thesis through CleanM and CleanDB opts for a declarative approach to data cleaning compared to the manual "cleaning by example" approach of the aforementioned systems. Also, CleanM exposes a superset of their functionality which is extensible to support more operations. In addition, whenever CleanDB requires additional information, e.g., the set of valid values of a domain, it employs auxiliary data structures such as dictionaries. Thus, the user is released from having to specify specialized and complex information.

(Semi-)Automatic Cleaning

Besides interactive cleaning toolkits, another category of systems attempt to detect and repair data errors automatically, asking a human for guidance when necessary.

Transformation tools: DataXFormer [7] is an automatic transformation tool that tackles semantic transformations, such as mapping a column that contains company names to a column with the stock symbols of those companies. Semantic transformations are challenging to perform because they require auxiliary information in order to infer the resulting value. For this, DataXFormer exploits additional information from the Web or from mapping tables. Transform-by-example (*TDE*) [58] allows semi-automatic search of transformations by synthesizing programs based on input transformation examples. To synthesize the programs, *TDE* exploits code libraries, web service APIs and web tables that it analyzes offline and uses them as an index to find the transformations.

Deduplication Tools: Tamr, the commercial version of Data Tamer [116], focuses on duplicate elimination using blocking. Specifically, Tamr uses classification to map records to corresponding blocks. Then, within each block, Tamr trains classifiers that decide whether a pair of records corresponds to a duplicate, a non-duplicate, or a possible duplicate. At the end, Tamr produces groups of values that represent the same entity and suggests a representative value per group using rules and auxiliary information by experts. The FUSE BY [21] extension of SQL is an operator which is used to resolve duplicate records by allowing various conflict resolution strategies, such as choosing the most common value or preferring one source

than another. FRAQL [112] follows a similar approach by providing an SQL extension that allows transformations, duplicate elimination and outlier detection. All conflict resolution operations in FRAQL are expressed in the form of UDFs. Tamr, FUSE BY and FRAQL focus on the repairing phase while CleanM focuses on the efficient detection of violations. Dedoop [71] allows the specification of entity resolution workflows, such as blocking, through a web-based interface and then translates them into MapReduce jobs. Ajax [48] generalizes the idea of Dedoop by separating the logical and physical level of the data cleaning process. Ajax allows the design of a data flow graph which represents the steps of a cleaning operation. Then, at the physical level, each logical operator gets translated into an optimized implementation. Ajax and Dedoop differentiate from CleanM in that they provide a UDF for each operator and therefore, treat the overall data cleaning process as a black box.

SampleClean [129] is a system that uses techniques from sample-based approximate query processing in the context of data cleaning. More specifically, SampleClean extracts a sample out of a dirty dataset, employs users to clean it and then utilizes this sample to answer aggregate queries. SampleClean focuses on transformations and deduplication operations.

Cleaning denial constraint violations: NADEEF [32] handles the case of denial constraints. NADEEF follows a systematic, holistic approach by managing a set of input rules as a whole. Specifically, users define a set of data quality rules through a programming interface; the system then performs violation detection by checking whether the rules hold for the given dataset. NADEEF then tries to update the values of the cells that are under violation in a way that all the rules are satisfied [27].

BigDansing [66] ports the insights of NADEEF in a distributed setting. Specifically, BigDansing extends MapReduce-like frameworks with support for duplicate elimination and denial constraints. BigDansing takes as input a dirty dataset along with a quality rule that is either declarative or has the form of a UDF. Then, BigDansing detects and repairs violations in a scale-out fashion. Since denial constraints often involve inequality joins, BigDansing provides a join operator that is designed to handle efficiently theta joins with inequality predicates. BigDansing focuses on physical-level optimizations, such as improving the degree of parallelism for a given cleaning task. CleanDB incorporates such low-level optimizations, but also allows for optimizations in all levels of the query translation process, such as language level simplification of data cleaning expressions. In addition, CleanM handles the scalability issue of data cleaning operations at the language level: every operation mapped to the monoid algebra is inherently parallelizable. Holoclean [103] repairs data by combining integrity constraints, master data, and quantitative statistics.

In summary, existing (semi-)automatic cleaning approaches optimize for a specific operation, such as transformations, integrity constraints, or duplicates. The approach followed in this thesis covers and optimizes those operations in a unified way, it is extensible to cover more operations, and it allows for incorporating optimizations at the physical level.

2.3 General-purpose data cleaning approaches

Quantitative Data Cleaning (QDC)

QDC [16, 34] 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 our approach in that it focuses on discovering the optimal repair method given a set of detected errors. CleanM focuses on the detection of the errors, whereas Daisy provides probabilistic repairs. 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 [17].

SQL for cleaning

SQL can express some cleaning tasks, e.g., the ones that correspond to first order logic statements [40]. 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 [29]. 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 [95] requires coding in an operator per algorithm, which is a tedious process. As for frameworks such as Spark [132], which support both relational and iterative processing, they apply only relational optimizations [14]. 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 [132]). 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 3.3, 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.

2.4 Analysis-aware data cleaning systems

The majority of the aforementioned tools operate offline before the analysis takes place. In the following, we survey systems that take into consideration the analysis.

`QuERy` [10] intermingles deduplication with query processing. `QuERy` uses blocking [94] for preprocessing and introduces operators in the query plan, which operate over the blocks. `QuERy` also optimizes the plan that involves cleaning operators. SampleClean [129] extracts a sample out of a dataset with duplicates and cell-level errors, asks users to clean it, and uses the sample to answer aggregate queries. SampleClean estimates the query result given the cleaned data and corrects the error of the queries over the uncleaned data. `QuERy` and SampleClean address entity resolution, duplicates, or cell-level errors, whereas Daisy focuses on integrity constraints. Also, `QuERy` differs in that it requires preprocessing to apply the blocking. Finally, SampleClean supports only aggregate queries. ActiveClean [74] incrementally updates a machine-learning model as the user cleans the data. ActiveClean addresses cell-level corruption cases, excluding cases that involve multiple records such as integrity constraints. DirtyLearn [98] follows a similar approach by performing relational learning over dirty data. Specifically, DirtyLearn extends relational learning algorithms by adding repair rules for matching dependencies and conditional functional dependencies. ImputeDB [24] considers query processing over data with missing values and decides whether to drop tuples by choosing the optimal solution in the efficiency/quality trade-off. ImputeDB also limits itself to cell-level errors.

The area of consistent query answering [13, 36, 70, 47] focuses on computing the query answers that are consistent with all possible repairs without modifying the data. This thesis through Daisy differs from this idea in that it computes all candidate qualifying tuples, and it applies repairing incrementally, driven by the queries and the denial constraints.

2.5 Graph pattern matching systems

In this section we survey related work on pattern matching over graphs. Graph processing systems can be classified into the think-like-vertex (TLV) approaches [86, 83, 105, 106], and the pattern matching systems [121, 60, 12, 45, 84] that follow a relational or a graph-based layout.

2.5.1 Think-like-vertex (TLV)

TLV systems focus on solving local neighborhood problems such as retrieving information of an entity, page rank, etc. GraphLab [83] is a vertex-centric shared-memory graph processing system that targets Machine Learning tasks and focuses on synchronization and scheduling of the iterative ML process. Pregel [86] is also a vertex-centric graph processing system that employs the Bulk Synchronous Parallel model to propagate a computation from a vertex to the neighbors. The pattern matching problem that we target differs from the idea of "think-like-vertex" as it involves processing over subgraphs instead of single vertices. Pattern matching has significantly higher complexity which grows exponentially with the pattern size, therefore it is harder to optimize.

2.5.2 Pattern matching

Pattern matching systems are divided based on the way they store, traverse, and partition the input graph. Specifically, there exist (a) breadth-first-search and relational approaches, (b) approaches that operate over a graph layout, (c) RDF-based approaches, and (d) workload-aware partitioning approaches.

Breadth-First-Search and relational approaches: Arabesque [121] is a distributed graph processing system that addresses the load-balancing problem in pattern matching. Arabesque introduces the notion of "think-like-embedding", where embedding is a subgraph of the graph. Arabesque iteratively repartitions and expands the discovered qualifying partial embeddings; it starts by uniformly partitioning subgraphs of a single vertex, then expands them to two vertices, repartitions the two-vertex subgraphs, etc. Even though the continuous repartitioning balances the load, it requires paying the cost of traversing and repartitioning the data after each vertex expansion. Our approach differs from Arabesque in that (a) it avoids traversing multiple times the neighbors of a vertex by indexing the qualifying ones, and (b) it addresses the load-imbalance problem without any extra cost, by using work-stealing. Neo4J [90] with the accompanying query language Cypher [31] and GraphX [53] represent graph data by using relational property tables, therefore they are inefficient as they store redundant information. GRAMI [37] follows a centralized approach for mining frequent patterns. GRAMI's optimizations focus on avoiding the materialization of all possible subgraphs as it only stores the templates of the frequent subgraphs, and in addition it stops when it computes the subgraphs that exceed the given threshold. Therefore, even though pattern matching is the building block of the frequent subgraphs problem that GRAMI addresses, its focus is on pruning the search space given that the interest is only on the frequent patterns.

Relational-based approaches for graph queries provide algorithms for worst-case optimal joins [11, 46, 8, 88]. Approaches such as EmptyHeaded [8] and [46] build indexes that ensure the optimality for joins with large intermediate results. BIGJoin [11] extends the idea in a parallel setting and employs a similar approach to Arabesque by having multiple rounds of parallel execution and synchronization while opting for minimal intermediate results and

balanced load. Such approaches require an expensive preprocessing step to build the index and have been shown to perform worse than binary joins for cases with fewer intermediate results [88, 46]. Our approach solves this problem by providing an intermediate solution that integrates the pruning of non-viable intermediate states while joining the partial states.

DFS subgraph isomorphism algorithms Ullmann's algorithm [124, 125] is a backtracking algorithm that tries to identify all possible subgraphs of a given graph and check whether they represent an isomorphism to the pattern. Ullmann's algorithm employs pruning mechanisms by allowing a qualifying mapping of the computed subgraph with the pattern in the case where the vertex of the graph is of at least the same degree with the vertex of the pattern.

VF2 [30] improves over Ullmann's approach in that it prunes non-candidate vertices. VF2 iterates through the vertices of the graph until it finds the first qualifying vertex. Then, it proceeds by selecting a matching vertex that is a neighbor of one of the vertices that already appear in the partial match. VF2 uses a set of pruning mechanisms to reduce the candidate vertices that it needs to check whether they can expand the partial match. The pruning mechanisms are called *feasibility criteria* and are based on a set of k-look-ahead rules for checking whether a partial match will have a non-qualifying successor after k-steps. In that case, it prunes the last added vertex and proceeds with another candidate. The most straightforward rule excludes vertices which are not connected with any of the vertices that belong to the partial match. Other rules are based on the structure of the graph and involve pruning a vertex if it does not lead to a complete match given its neighbors.

More recent approaches that follow a graph-based layout optimize the filtering as well as load balancing of pattern matching [76, 118]. GraphQL [57] proposes a query language for graphs. GraphQL also introduces local and global pruning techniques to reduce the search space of subgraph isomorphism while it also optimizes the search order of the nodes. TurboISO [56] and its extension BoostISO [104] identify candidate regions to explore for patterns and optimize the order of evaluation. Further extensions of TurboISO consider decomposing the query graph into dense and sparse areas to accelerate pattern evaluation, while in addition, they estimate the cardinality of the embeddings to further optimize the matching order [19]. CECI [18] first builds a compact embedding cluster index using BFS. The cluster index stores candidate matches by preserving the connectivity of the query graph. By using the index, CECI applies lightweight filtering to prune non-candidate nodes and allow faster pattern matching. The cluster index also allows for parallel execution by assigning the clusters to different workers. CECI requires however expensive preprocessing by checking for cluster overlap to avoid duplicate work by different workers. Our approach uses and extends the filtering techniques of existing subgraph isomorphism approaches in a parallel setting. The main difference is that we propose techniques tailored to a scale-up scenarios by avoiding preprocessing in order to (a) prune intermediate results, (b) eliminate redundant work, and (c) balance the load.

RDF-based systems SPARQL queries over RDF data translate into matching patterns over

RDF graphs. Therefore the problem of SPARQL query evaluation resembles the graph pattern matching problem. RDF systems such as the Virtuoso RDF system [38] sotre RDF data in a relational table and translate SPARQL queries to SQL. By this, they allow the database optimizer to optimize query execution. Similarly, CliqueSquare [52] employs and optimizes $n-$ary star joins as bushy plans to increase parallelization. Both approaches require flattening graph data, and in addition Virtuoso requires executing many self-joins to compute qualifying paths. RDF3X [91] optimized for SPARQL queries, materializes in relational tables all possible subject-predicate-object subsets. Unlike RDF, in the case of general graphs materializing relationships is impossible. What is more, the materialization is workload agnostic, thereby requires storing tuples that do not lead to a qualifying pattern.

Partitioning approaches: *TAPER* [45] proposes a partitioning technique that minimizes inter-partition traversals for a given set of path queries over a labeled graph. *TAPER* starts from an existing partitioning and incrementally updates the partitioning until it achieves an optimal partitioning for the given known workload. To reduce the inter-partition traversals given the workload, *TAPER* employs the idea of visitor matrix which encodes the probability of transition from one edge to the other. In contrast to our approach *TAPER* addresses only the partitioning problem, it assumes a known workload, and the visitor matrix that it maintains has size which is orders of magnitude larger than the original graph. [84] extends the idea of *TAPER* to support arbitrary patterns. To estimate the inter-partition accesses, it estimates the edge traversal probability for a given workload. The difference between the aforementioned existing workload-aware approaches is that both *TAPER* [45] and [84] require prior knowledge of the workload and they also involve expensive preprocessing in order to balance the load.

3 Unified Scale-out Data cleaning

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 [81]. 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 [62, 101]. 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 [32, 66]. 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 that depends on the specific needs of the data hence requires a language that enables 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 thesis 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* [44]. 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, 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 [61] – 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 [14] 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 thesis 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.

3.1 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 [9, 63], 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.

3.1.1 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* [40] 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 [72]. 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 [59].

Transformations & Term Validation

Transformations involve applying a formula to a set of values, or mapping values to a set of semantically related values [7]. 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.

3.1.2 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* [44]. 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 [65, 64]. 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 [23, 43]. Section 3.2 elaborates on how cleaning operations are mapped to CleanM.

The second abstraction level involves lowering a comprehension into an algebraic form [44], 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 3.3 discusses the algebraic rewrites.

The final level specializes the algebraic expression to the underlying execution engine. CleanM currently assumes that Spark [132] 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 3.4 discusses how to generate physical plans that consider both these complications.

3.2 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 .

3.2.1 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 $\mathscr{U}_{\oplus}$ 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. We use Scala-like comprehension syntax which is equivalent to the one presented, representing a comprehension as $for\{q_1, ..., q_n\}\, yield \oplus e$.

3.2.2 Optimizations at the monoid level

As discussed in Section 3.1.2, CleanM follows a layered design approach. Even in its top-most 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 3.1.1. Both high- and low-level operations go through a rewrite process that applies general-purpose, domain-agnostic optimizations [44].

Domain-agnostic optimizations: Normalization

Regardless of the processing that a comprehension performs, a normalization algorithm [44] 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 [68]. 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 3.3 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 3.1.1 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 3.1.1 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 3.4 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 [61, 111], 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.

3.2.3 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 [44]. Also, monoid comprehensions are sufficient to represent OQL and SQL queries [44]. 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 [111]. 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 opera-

tions; the latter step is the one performing the actual clustering/partitioning.

We express *center initialization* by parameterizing *the function composition monoid* [44] 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 [128], 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 [44].

Multi-pass partitional algorithms. Representing multi-pass partitional algorithms (e.g., the original k-means, canopy clustering [87], correlation clustering [115], 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. The number of iterations can be either given as input, or a stopping condition can be integrated in the iteration process.

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 [54] 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 words into clusters of similar contents (e.g., filtering based on the length of the words [54, 110, 78]). Similarly, locality-sensitive-hashing techniques [51] follow the same principle by creating buckets of values based on a set of hash functions and then the comparison takes place among values that have the same or overlapping signatures. Other filtering approaches such as applying transitive closure in order to build the similar pairs can be also represented using the monoid calculus. CleanM can also incorporate similarity groupings [120] that operate over high-dimensional data by representing them as associative operations. Such similarity operators may appear with different semantics, therefore CleanM is a natural fit for expressing them; CleanM can represent set membership operations based on correlation metrics, and then it can resolve group overlaps through the merge phase.

Apart from denial constraints that constitute a popular subcategory of constraint-generating dependencies, CleanM can support also tuple-generating dependency checks, such as foreign-key constraints. Tuple-generating dependencies are of the form:

$$\forall x_1..x_n \phi(x_1, ..., x_n) \rightarrow \exists z_1...\exists z_k \psi(y_1, ..., y_m)$$

where ϕ, ψ are relational atoms. Such tuple-generating dependencies are possible by representing them as existential quantifications that the monoid calculus allows. Finally, operations such as outlier detection [82] are also possible since they can be detected by using statistics which can also be represented as monoid comprehensions [23].

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.

```
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.

3.2.4 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 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.

3.3 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*

Operator Name	Select	(Outer) Join	Reduce	Nest	(Outer) Unnest
Operator Symbol	$\sigma_p(X)$	$X ⟕_p Y$ $X ⟗_p Y$	$\Delta_p^{\oplus/e}$	$\Gamma_p^{\oplus/e/f}$	$⊥\!\!\!\mu_p^{path}(X)$ $\mu_p^{path}(X)$
Superscript & Subscript	p: Filtering Expression f: Groupby Expression $\oplus$: Output Type / Monoid			e: Output Expression $path$: Field to unnest	

Table 3.1 – The operators of the nested relational algebra.

relational algebra expression [44], 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 [44]; the result is a logical plan that uses the operators of Table 3.1. *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 [44]. 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 3.1 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 [44]. 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 3.1. Plan A performs term validation via token filtering: It unnests the list of names

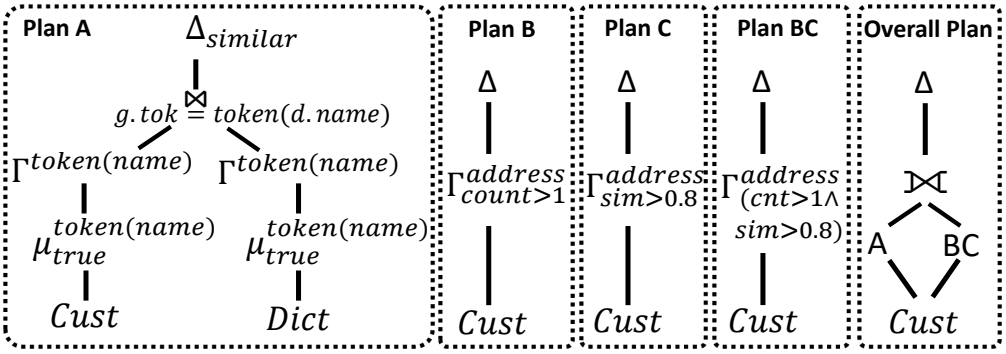


Figure 3.1 – Algebraic plans for our running example, and optimized rewritten plans that coalesce operators and share work.

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 $(token_i, \{names\})$ for each element of a nested collection to be processed; it operates over smaller $(token_i, name_j)$ tuples instead [65]. 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.

3.4 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 [132] 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 [14], 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-

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)))
$=\mu_p^{path}$	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
$=\bowtie_{f(A)=g(B)}$	left outer join
$=\bowtie_{f(A)\ \theta\ g(B)}$	**theta join** → map

Table 3.2 – Translation of algebraic operators to Spark operators. Bold parts introduce new Spark operators or deviate from the translation that Spark SQL would have performed.

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 3.2 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 [92].

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

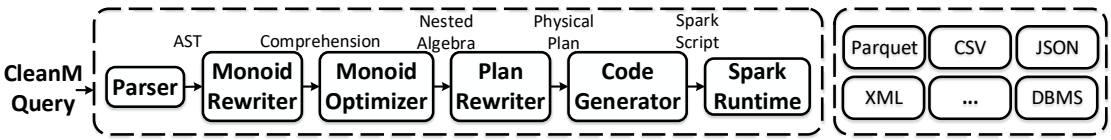


Figure 3.2 – The architecture of CleanDB.

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 3.2 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 [92]. 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.

3.5 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 [63], an adaptive query engine that operates over raw data. Specifically, we extend the commercial version of RAW [102], which operates over the Spark runtime. CleanDB serves as a replacement layer of Spark SQL [14]; it exposes the expressive power of CleanM without the compromises that 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 3.2 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

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 3.3 – Accuracy of term validation approaches over the DBLP dataset.

the monoids presented in Section 3.2. 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 [73]. 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 [49].

3.6 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] [66] 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

[1]We thank the authors of [66] for giving us access to a binary executable.
[2]SampleClean [129] only operates over query-specific samples.

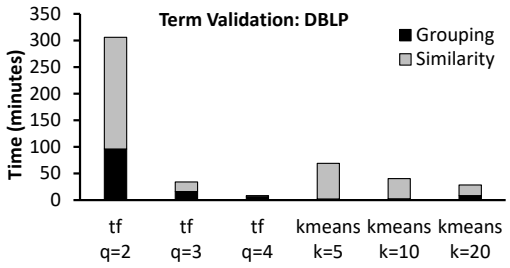


Figure 3.3 – Different configurations of term validation.

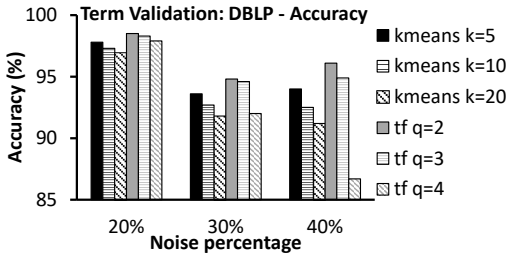


Figure 3.4 – Accuracy of term validation as the noise increases.

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 × 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 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) [114], 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.

3.6.1 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 3.3 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 3.3 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 3.4 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 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.

3.6.2 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.

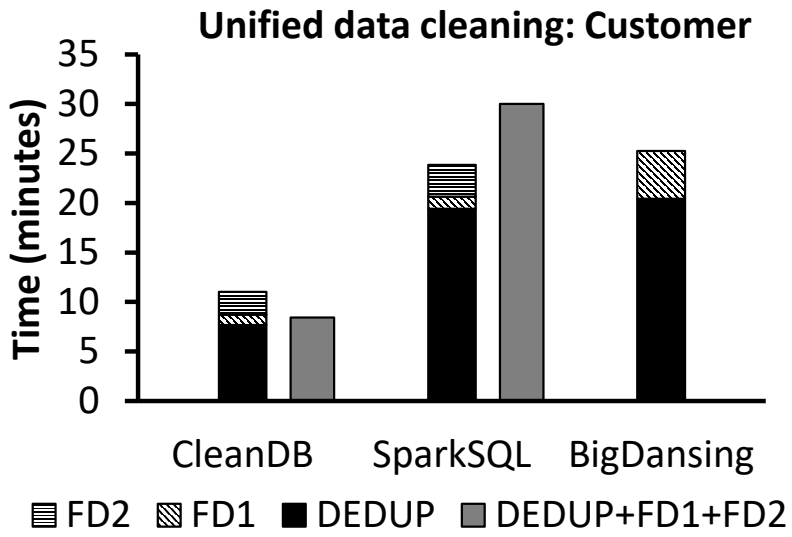


Figure 3.5 – CleanDB rewrites three cleaning operations into a single one, and avoids duplicate work.

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 3.5 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 3.4 – Overhead introduced by performing syntactic transformations in a plain query. The optimizer of CleanDB applies both operations in one go and reduces overhead by ~ 2×.

Results. Table 3.4 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.

3.6.3 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%.

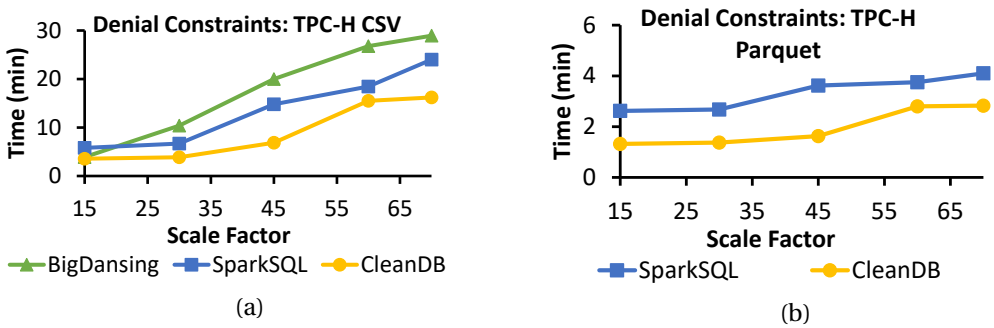


Figure 3.6 – Cost of checking for violations of functional dependencies over TPC-H.

$\phi : orderkey, linenumber \rightarrow suppkey$ 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 3.6a, 3.6b 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 3.6a) and Parquet (Figure 3.6b). Parquet is only supported by CleanDB and Spark SQL; we omit BigDansing in Figure 3.6b. The response times of Figure 3.6b are shorter than those of Figure 3.6a 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 [130]: 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.

DC Results. The detection of violations of rule ψ involves a self-join that checks the inequality

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

Table 3.5 – Denial constraints involving inequalities as the dataset size increases. All systems beside CleanDB fail to terminate.

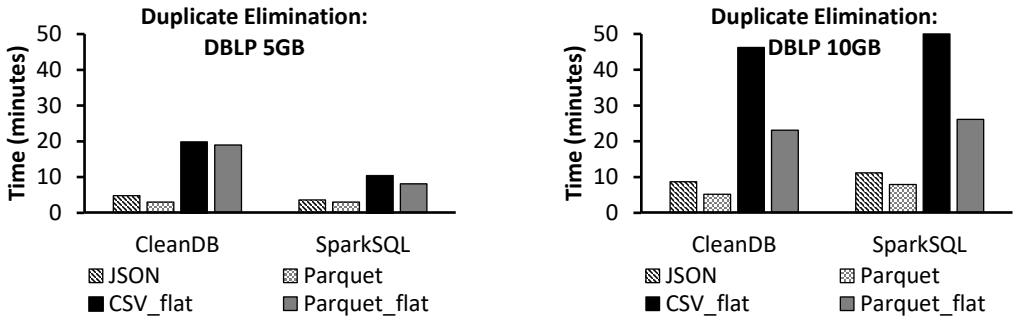


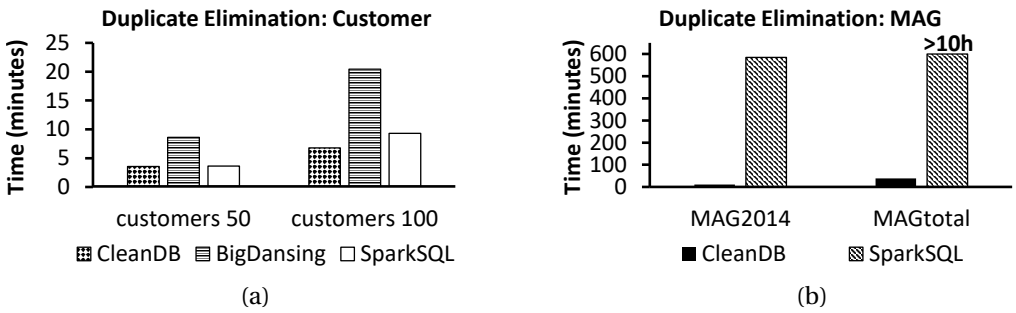
Figure 3.7 – Duplicate elimination over different representations of DBLP. We simplified the dataset for Spark SQL to terminate.

conditions. Table 3.5 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 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



(a) (b)

Figure 3.8 – Duplicate elimination over Customer and MAG.

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 3.7 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 3.8a 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 3.8b 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.

3.7 Summary

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.

4 Cleaning denial-constraint violations through Relaxation

4.1 Introduction

Real-life data contain erroneous information, which leads to inaccurate data analysis [55, 74]. Data scientists spend most of their time cleaning their data [81], until they are able to extract insights. Depending on the accuracy requirements of the workload and the data they need to access, users iteratively apply cleaning tasks until they are satisfied with the resulting quality. Thus, data cleaning is a time-consuming process.

Data cleaning is an interactive and exploratory process that involves expensive operations. Error detection requires multiple pairwise comparisons to check the satisfiability of the rules [50]. Data repairing adds an extra overhead as it requires many iterations of assigning candidate values to dirty cells, until all rules are satisfied [27, 32]. Data scientists also detect inconsistencies and constraints at data exploration time [10]. Hence, traversing the whole dataset multiple times to repair each discovered discrepancy is cost-prohibitive.

State-of-the-art approaches can be divided into offline, and online analysis-aware approaches. Offline tools [32, 66, 103] treat data cleaning as a separate process, decoupled from analysis. Applying data cleaning before analysis begins requires prior knowledge of the errors that exist. Offline cleaning is also cost-prohibitive, as it operates over the whole dataset [40]. Analysis-aware tools [10, 129, 74, 24] focus on entity resolution or deduplication, or they limit themselves to cell-level errors. But entity resolution tools either require expensive preprocessing [10] or support only aggregate queries [129].

Besides the need for online, real-time cleaning, the repair of errors requires human effort [85]. Accurate repair requires master data, which in real-world scenarios are either not available or are insufficient for the domain-specific datasets that users work with. Therefore, machine learning based cleaning solutions that automatically apply fixes of dirty values based on general-purpose training data that stem from a large corpus fail to provide accurate repairs [25].

Name	Zip	City
Jon	9001	Los Angeles
Jim	9001	San Francisco
Mary	10001	New York
Jane	10002	New York

Table 4.1 – Employees dataset.

There is a need for an efficient, probabilistic cleaning approach that is weaved into the exploratory analysis and that cleans data on-demand. On-the-fly cleaning repairs only necessary data, thus if only a subset is analyzed, the wasted-effort is minimized. Online cleaning also benefits offline cleaning by enhancing the predictability on the required cleaning tasks. By providing probabilistic candidate fixes, users can interactively repair their data by consulting the proposed values. Thus, integrating probabilistic cleaning with analysis efficiently supports exploratory applications [33] by reducing data-to-insight time.

Example 1. *Consider the dataset of Table 4.1 that comprises employees information. Assume that a user is interested in analyzing all employees in Los Angeles. The insights that the user extracts might be incorrect due to the conflict among the first two tuples that have the same zip code and different city name; they violate the functional dependency zip→city stating that the zip code defines the city. Hence, the analysis ignores the second tuple whose city is San Francisco, but after cleaning it, it might obtain the value Los Angeles. Having to clean the whole dataset is unnecessary as (i) the user is interested only in a subset of the data, and (ii) the query result can be cleaned by checking only the relevant data subset.*

We present the first approach that intermingles cleaning denial constraint violations with exploratory SPJ (Select-Project-Join) and aggregate queries, and that gradually cleans the data. Denial constraints (DCs) comprise a family of rules that have been widely used to capture real-life data inconsistencies [108, 40]. To provide correct results over dirty data, we introduce cleaning operators in the query plan and employ a cost model to optimally place them. To enable cleaning operators to detect errors, we define at the execution level a novel query-result relaxation mechanism in the context of DCs. Query-result relaxation enhances the query result with correlated data from the dataset to allow error detection. Then, given the detected errors, we propose candidate fixes by providing probabilistic results [117]. We validate our approach by building Daisy, a distributed incremental cleaning framework over Spark [132].

Contributions: Our contributions are as follows:

- We present a query result relaxation mechanism that enables interleaving SPJ and aggregate queries with cleaning DC violations. Our approach guarantees correctness, compared to the offline approach in the case of functional dependencies, and provides accuracy estimates in the presence of general DCs.

- We introduce cleaning operators inside the query plan by using a cost model that determines the execution order of cleaning and query operators at query time.

- We implement Daisy, the first system that enables exploratory data-analysis queries over data with DC violations. We execute Daisy over Spark, and experimentally show that it is faster than offline cleaning solutions on synthetic data and supports real-world workloads that offline cleaning fails to address.

4.2 From Offline to Online Data cleaning

Problem Statement: We need to efficiently clean in real-time exploratory query results in the presence of dirty data. We clean denial constraint (DC) violations [40] as they involve a wide range of rules that detect semantic inconsistencies in the data. DCs are universally quantified first-order logic sentences that represent data dependencies, including functional dependencies (FDs). DCs are defined as: $\forall t_1, ..., t_k \neg(p_1 \wedge p_2 ... \wedge p_m)$, where each t_i is a tuple, each p_i is a predicate involving conditions between the attributes of one or more tuples, k is the number of involved tuples, and m is the number of predicates.

Challenges: Interleaving cleaning with querying must provide accurate results, without cleaning the whole dataset. Moreover, as cleaning is costly compared to query processing, adding the cleaning overhead over each query might result in an overall cost higher than offline cleaning. Also, during data exploration, users have partial knowledge on the rules that hold; cleaning a value, given partial knowledge affects the resulting data quality [16]. Even when the rules are known, automatically fixing an error might result in inaccuracies [103]; human effort or master data are required.

Solution: To efficiently and accurately provide correct query results in the presence of DCs, we weave cleaning operators into the query plan. We optimize the overall execution by detecting the relevant data subset that affects the cleanliness of the result, and we introduce a cost model to optimally place the cleaning operators, based on the overhead they add on each query. To capture partial knowledge of the rules and the data, we clean data by providing probabilistic fixes. Then, using our solution once all rules are known and given the probabilistic suggestions, we can either use inference [131, 103, 80] when master data exist, or have humans fix the errors in the query results. Inference approaches over the probabilistic data are complementary and out of the scope of this work. Future work includes examining the human cost of cleaning the flagged dirty values of the query results.

4.3 Query Execution over Dirty Data

Executing queries over dirty data induce wrong query results [74]. A tuple might erroneously appear or be missing from a query result due to a dirty value. We describe how we fix wrong query-results, by detecting and cleaning potentially qualifying tuples. To detect qualifying

tuples, we introduce the query-result relaxation mechanism that relaxes results, based on the dependencies defined by the rules. Query result relaxation differs from query relaxation [113] in that instead of relaxing the conditions of the query, it relaxes the result. Still, we relax the result to repair conflicting tuples based on the input DCs, whereas query relaxation is used to deal with failing queries and incomplete databases. The relaxation process takes place in the case where the tuples belonging to the query result participate in a violation. To clean the relaxed result, we provide probabilistic fixes based on the frequency of each candidate value in a dirty cell. Then, we update the dataset in-place with the probabilities. We also maintain provenance to the original values in case new rules appear. Thus, we gradually transform the dataset into a probabilistic dataset.

Our probabilistic representation uses attribute-level uncertainty [117]; attributes take multiple candidate values. To represent candidate tuples (i.e., possible worlds) by using attribute-level representation, we store in each candidate value an identifier of the possible world it belongs to. To compute the probabilistic fixes we assume a correlation between the attributes of the dataset; data repair that exploits attribute correlation is the state-of-the-art approach for fixing dirty values assuming no master data exist [131, 103].

Then, query operators output a tuple iff at least one candidate value qualifies. Thus, (self-)joins on probabilistic join-keys output a pair iff the candidate values of the join-keys overlap. To enable reasoning over the data, each tuple of the result contains all candidate values. For (self-)joins, we also employ a similar approach to the lineage used in probabilistic data [117]; we store in the result the originating tuple IDs because if a potential inference updates a join key value, a pair might no longer satisfy the join. In the following, we introduce cleaning operators that enable cleaning at query time.

Definition 1. *A cleaning operator is an update operator that receives a query-result or a relation and outputs the clean result or relation. When the cleaning operator takes input from a query operator it (a) relaxes the result based on the dependencies of the input DCs, (b) detects and fixes errors, and (c) updates the data in-place with the clean values.*

Cleaning operators differentiate between Select and Join operators. For group-by queries, cleaning takes place before the aggregation; to avoid grouping recomputation, we push down cleaning either over any underlying select or join condition, or over the input relation. Below, we present our probabilistic cleaning approach for SP and SPJ queries, given one or more DCs using relaxation.

4.3.1 Cleaning SP query results given a FD

Definition 2. $clean_\sigma$ *is a cleaning operator that relaxes and cleans the result of a select operator.*

The first step of $clean_\sigma$ is to relax the result. Consider a dataset with schema S, and a FD ϕ: $X \rightarrow Y$, where $X \subseteq S$, $Y \subseteq S$. X might contain multiple attributes, whereas Y contains one

Algorithm 1: SP query result relaxation for FDs

```
   input: Dataset d, Query answer A, FD(lhs,rhs)
1  total_extra = A
2  unvisited = d - A
3  while total_extra ≠ ∅ do
4      A_lhs = total_extra.map(x → x_lhs)
5      A_rhs = total_extra.map(x → x_rhs)
6      extra = unvisited.filter(x → A_lhs.contains(x_lhs))
7      total_extra = total_extra ∪ extra unvisited = unvisited - extra
8      extra = unvisited.filter(x → A_rhs.contains(x_rhs))
9      unvisited = unvisited - extra
10     total_extra = total_extra ∪ extra
11 return total_extra
```

attribute; if Y contained more attributes (e.g., Y_1, Y_2), then ϕ would be mapped to multiple FDs (e.g., ϕ_1:$X \rightarrow Y_1$, ϕ_2:$X \rightarrow Y_2$) [59]. Given a SP query with projection list $P \subseteq S$, and where clause attributes W⊆S, ϕ affects query correctness iff (X∪Y)∩ (P∪W) ≠∅, i.e., iff the query accesses an attribute of ϕ. If the query overlaps with ϕ, $clean_\sigma$ augments the result with tuples from the dataset that have the same value for X and/or Y. We refer to the extra tuples as correlated tuples.

Algorithm 1 shows the general query result relaxation that uses transitive closure; it iteratively computes the correlated tuples of the result, until no more correlated tuples are detected. Consider an FD $lhs \rightarrow rhs$, and A_{lhs}, A_{rhs} being the set of left-hand-side (*lhs*) and right-hand-side (*rhs*) attribute values that appear in the result (lines 4,5). Algorithm 1 traverses the data subset that does not belong to the relaxed result (unvisited) (lines 2,9) and enhances the result with each tuple x for which $\{x_{lhs} \in A_{lhs}\}$ or $\{x_{rhs} \in A_{rhs}\}$ (lines 6-10).

The second step of $clean_\sigma$ is to detect errors and compute fixes given the relaxed result. Consider random variables *LHS*, *RHS* that represent the candidate *lhs* and *rhs* values of an erroneous tuple *t*. *LHS* contains the *lhs* values of the tuples t' for which t'_{rhs}=t_{rhs}, i.e., they have the same *rhs* value. *RHS* contains the *rhs* values of the tuples t' for which t'_{lhs} =t_{lhs}. Hence, by including all correlated values, future accesses to the cleaned tuples require no extra checks. Candidates $c_{lhs} \in LHS$, $c_{rhs} \in RHS$ have probabilities P($c_{lhs}|t_{rhs}$), P($c_{rhs}|t_{lhs}$), respectively. Thus, based on attribute dependencies, each tuple can have two instances, one with the candidate c_{lhs}, given the existing t_{rhs} and one with the candidate c_{rhs}, given the existing t_{lhs}. As in our internal representation we use attribute-level uncertainty, we store inside each candidate value the ID of the candidate pair it belongs.

As Algorithm 1 is iterative, we need to estimate the number of iterations required, as well as the relaxed result size, to accurately compute the fixes using the correlated tuples.

Lemma 1. *Algorithm 1 requires one iteration to enable accurate candidate fixes in the presence of SP queries with a filter on the rhs of an FD.*

Proof. Consider a query with a filter restricting the *rhs* over the range *[a,b]*. The correct result

Zip	City
9001	Los Angeles
9001	San Francisco
9001	Los Angeles
10001	San Francisco
10001	New York

(a)

Zip	City
9001	Los Angeles, 67% San Francisco, 33%
9001	Los Angeles, 67% San Francisco, 33%
9001 50% 10001 50%	San Francisco
9001	Los Angeles, 67% San Francisco, 33%
10001	San Francisco
10001	New York

(b)

Table 4.2 – Cities dataset: (a) Dirty version, (b) Partially clean version with candidate values. The dashed line denotes different candidate fixes for the erroneous tuples.

must include both the clean tuples with rhs values in the range *[a,b]*, as well as the dirty tuples that are candidates to take values in *[a,b]*. Algorithm 1 computes the tuples that have matching *lhs* with the dirty result (line 6). We assume that, to exploit the dependency and be able to make a prediction, the dirty tuples contain either a clean *lhs*, or a clean *rhs* [131]. The extra tuples with matching *lhs* are the candidates to get *rhs* in *[a,b]*. Hence, enhancing the result with tuples having the same *lhs* guarantees no missing tuples. We also show that the included tuples contain all candidate values. Algorithm 1 covers the *rhs* candidate values by computing the tuples with matching *lhs*. Then, Algorithm 1 computes the *lhs* values of all tuples with matching *rhs*. However, these tuples are already included in the enhanced-result as they satisfy the query; thus the algorithm terminates. □

Example 2. *Consider the dataset of Table 4.2a, the FD $Zip \rightarrow City$, and a query requesting the zip code of "Los Angeles".*

The dirty result consists of the first and the third tuple of Table 4.2a. $clean_\sigma$, by following Algorithm 1, enhances the result with the tuples that have the same *lhs* with the result, that is the tuples for which *{City≠Los Angeles∧Zip=9001}={9001, San Francisco}*. Afterwards, it adds the tuples of the set *{Zip≠9001∧ City=Los Angeles}=∅*, that is the ones that share a *rhs* value with the result. However, based on the proof of Lemma1, this set is empty since the tuples with *City=Los Angeles* already appear in the result. Then, $clean_\sigma$ computes the candidate fixes *LHS* and *RHS*, and their candidate probabilities *P(City|Zip)* and *P(Zip|City)* for the tuples of the updated result. For the first and the third tuple, *LHS* consists of the candidate values of the tuples t' that have t'_{rhs} =*Los Angeles*. Similarly, the *RHS* consists of the candidate values of the tuples that have t'_{lhs} = 9001, that is *San Fransisco, Los Angeles*. The corresponding probabilities of each value are given by the conditional probabilities *P(City|Zip=9001)* and *P(Zip|City=Los Angeles)*. For the second tuple there are two candidate pairs distinguished by a dashed line in the table for simplicity: *{City|Zip=9001}= {San Francisco 33%, Los Angeles 67%}* and *{Zip|City=San Francisco} = {9001 50%, 10001 50%}*. The updated version of the dataset is

Zip	City
9001	Los Angeles, 67% San Francisco, 33%
9001	Los Angeles, 67% San Francisco, 33%
9001 50%, 10001 50%	San Francisco
9001	Los Angeles, 67% San Francisco, 33%
10001	San Francisco, 50% New York, 50%
9001 50%, 10001 50%	San Francisco

Table 4.3 – Correct query result given condition on the *lhs*. The query result becomes accurate after traversing the dataset again to fetch more correlated entities.

shown in Table 4.2b.

A filter over the lhs requires multiple iterations in order to also include and fix dirty tuples that qualify the query as well as their context.

Example 3. *Consider the dataset of Table 4.2a and a query requesting the city name with zip code "9001".*

The dirty result comprises the first three tuples of Table 4.2b. However, given the conflict between the tuples with zip code 10001, the correct result contains the four tuples shown in Table 4.3. The fourth tuple qualifies because it has two worlds (*{{90001 50%, 10001 50%}, {10001}}*), and the first one satisfies the condition. Thus, Algorithm 1 adds the tuple *{10001, San Francisco}* since it contains a rhs value which appears in the result. Then, the next iteration adds the tuple *{10001, New York}* since *10001* belongs to the relaxed result. Thus, using transitive closure, Algorithm 1 determines the whole cluster of correlated entities.

Lemma 2. *Consider a query with a filter on the lhs, and a relaxed result A_R with maximal size $|A_R|$ at iteration i. Algorithm 1 requires an extra iteration to compute the candidate values with probability $Pr(\geq 1) = 1 - \binom{\#vio}{0}\binom{n-\#vio}{|A_R|}/\binom{n}{|A_R|}$, where Pr is the hypergeometric distribution, n the dataset size and the dataset has $\#vio$ violations.*

Proof. Algorithm 1 terminates when the computed augmented result contains no more errors, that is there are no tuples with the same *lhs* and different *rhs*. Consider iteration i, where the relaxed answer A_R has maximal result size $|A_R|$. The probability that A_R contains at least one violation is equivalent to the complement of the probability of having no violations $Pr(0)$. Using the hypergeometric distribution, we estimate $Pr(0)$ over the subset A_R, given a total population of size n that contains $\#vio$ violations. Thus, $Pr(\geq 1) = 1 - \binom{\#vio}{0}\binom{n-\#vio}{|A_R|}/\binom{n}{|A_R|}$. □

Thus, while cleaning the qualifying result, Algorithm 1 might also detect and repair extra violations of the correlated tuples of the result.

Lemma 3. *Let $\mathscr{A}$ be the set of attributes that appear in the FDs. Let c_i be the cardinality (number of distinct values) of each attribute $A_i \in \mathscr{A}$ in the query result, and D_i, Dq_i the frequency*

distributions of each A_i over the dataset and the query result respectively. The upper bound of the relaxed result size in each iteration is $\mathscr{R} = \sum_{i=1}^{|\mathscr{A}|}(\sum_{j=1}^{j=c_i} D_{ij} - \sum_{j=1}^{j=c_i} Dq_{ij})$.

Proof. Given that the c_i values of the result follow a distribution Dq_i, then the total frequency of these values over the dataset is $\sum_{j=1}^{j=c_i} D_{ij}$. The upper bound corresponds to the worst case scenario where there is no overlap between the sets of correlated tuples stemming from each attribute A_i. In the worst case, the number of extra tuples that the relaxation adds to the result set corresponds to the number of tuples sharing the same value for each attribute of the result. Therefore, in total, the number of tuples is: $\mathscr{R} = \sum_{i=1}^{|\mathscr{A}|}(\sum_{j=1}^{j=c_i} D_{ij} - \sum_{j=1}^{j=c_i} Dq_{ij})$. □

Thus, in the case of queries restricting the rhs, the upper bound is equivalent to $\mathscr{R}$.

To avoid the unnecessary cost of the relaxation in the parts of the dataset where no errors exist, we trigger Algorithm 1 only when the resulting query tuples participate in a violation. To compute the violating tuples, we precompute the group by based on the *lhs* of the functional dependency and count the number of different *rhs* values. Then we store the distinct *lhs* values that are erroneous. By this, when any of the resulting tuples contains a *lhs* value belonging to the erroneous values, then the relaxation takes place.

Relaxation benefit: The extra tuples contain the pruned domain of values that a system, or a user needs to infer the correct value of an erroneous cell [103]. Specifically, a query result contains a set of tuples with a restricted set of values for the attributes of the constraints. The cleaning process can exploit this characteristic and extract all the correlated tuples of the result, instead of computing the candidate fixes separately for each violated tuple. Thus, instead of traversing the whole dataset for each erroneous value to compute the candidate fixes, relaxation iterates over the correlated tuples.

4.3.2 Cleaning SP query results given a DC

We present the case of more general rules with arbitrary predicates. $clean_\sigma$ first computes the correlated tuples that, in the case of DCs, involve the conflicting tuples with the query result. Detecting the correlated tuples requires a self theta-join. We adopt an optimized parallel theta-join approach [92] that maps the cartesian product to a matrix that it partitions into p uniform partitions. Using the matrix, we check arbitrary predicates between any pair of attributes. In our analysis, we focus on the more realistic case that involves conditions over the same attribute [66]. We compute incrementally the theta-join, by partitioning and checking the matrix subset that affects the result; the matrix subset involves the query result and the unseen part of the dataset. We also prune the redundant symmetric parts of the matrix.

Partial theta-join operates at a finer granularity, hence it can prune a) matrix partitions and b) pairs within a partition, which are not candidates for conflicts. First, by partitioning a subset of the matrix, partitions have boundary ranges smaller than the more general boundaries

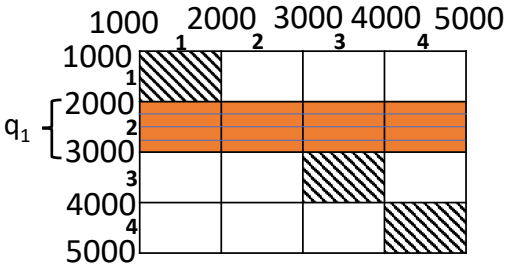


Figure 4.1 – Cleaning q_1

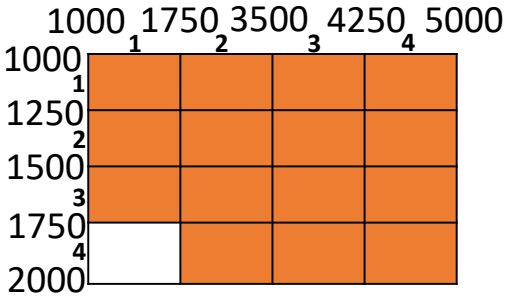


Figure 4.2 – Cleaning q_2

of the original matrix partitions; there might exist sub-partitions whose boundaries do not qualify the condition, even though the general partition qualifies. Second, partial theta-join prunes non-qualifying intra-partition pairs; within a partition, it restricts the candidate pairs to be checked.

Example 4. *Consider a dataset with salary, tax values and rule* ϕ:$\forall t_1,t_2$:$\neg(t_1.salary<t_2.salary \wedge t_1.tax>t_2.tax)$. *Fig. 4.1 shows an example cartesian product matrix based on rule* ϕ. *Consider two queries requesting salary ranges* [2000 – 3000] *and* [1000 – 2000] *respectively.*

To clean the first query result, theta-join checks for violations in the orange area of Fig. 4.1 which it divides into p partitions. Then, for the second query it constructs a matrix with vertical range (1000-5000)$\setminus$(2000-3000) since the subset (2000-3000) has been already checked, thus it excludes it from the comparisons. Given a smaller range, the boundaries will be the ones of Fig. 4.2. Theta-join can then filter out non-qualifying partitions, such as partition (4,1). It also applies intra-partition filtering to exclude non-qualifying pairs. For example, given partition (3,1) with horizontal and vertical ranges (1500, 1750) and (1000, 1750) respectively, since we are interested in checking the < condition, the vertical range is transformed into (1500, 1750) since the part (1000, 1500) will not produce candidate violations.

The second step involves computing the candidate fixes. For DCs we use the holistic data cleaning approach [27] to calculate the possible conditions that the dirty cells must satisfy. Specifically, given a rule with inequality predicates, $clean_\sigma$ replaces the errors with the candidate ranges that satisfy the constraints. Then, similarly to FDs, the probability of each candidate is frequency-based, given the total number of fixes.

More formally, given a DC $\forall t_i,t_j \neg(t_i.v_1>t_j.v_2)$ and two tuples t_1,t_2 for which $t_1.v_1>t_2.v_2$, then a candidate fix of the violation needs to enforce the constraint; $t_1.v_1=\{t_1.v_1 \ or <t_2.v_2\}$, $t_2.v_2=\{t_2.v_2 \ or >t_1.v_1\}$. Thus, each attribute value will either maintain its original value, or will obtain a value satisfying the range. In the case of DCs that contain more atoms, we map the dirty formula involving the conditions of the conflicting tuples to a SAT formula [32], where a subset of atoms must become false (invert their condition) in order to satisfy the formula. Thus, a possible violation fix requires updating the appropriate attribute values in order to invert the condition of the subset of atoms that cause the violation. Thus, to include

Algorithm 2: Inequality join selectivity estimation

```
1 ranges = split(d,p) for r1 in ranges do
2     for r2 in ranges do
3         if overlap(r1, r2) then
4             range_vio(r1) = count_overlap(r1, r2, r)
5 return range_vio
```

all the possible combinations of violated atoms, we produce all possible candidate attribute combinations. Then, a SAT solver [20] can decide on which atoms must remain true or need to invert their conditions (become false) in order to satisfy the whole DC formula.

The probabilities of each candidate fix are based on the frequency that each of the candidate ranges appears. We provide frequency-based probabilities to collect all possible fixes for a specific value, accompanied by their weight. Then, after having computed the candidate fixes of the data subset that affects the cleanliness of a value v_i, an inference algorithm can repair the dirty values. Future work considers updating the probabilities after accessing more data, thereby incrementally inferring the correct value.

Example 5. *Consider a dataset with {salary,tax,age} values t_1: {sal:1000,tax:0.1,age:31}, t_2:{sal:3000,tax:0.2,age:32}, t_3:{sal:2000, tax:0.3,age:43} and ϕ:$\forall t_1,t_2$:$\neg$(t_1.salary<t_2.salary$\wedge$ t_1.tax>t_2.tax).*

Tuples t_2,t_3 violate ϕ, thus the candidate fixes for t_2 are {(<2000 50%,3000 50%),0.2,32},{3000,(0.2 50%,>0.3 50%), 32}, that is, t_2 must either take a salary less than 2000 or have tax greater than 0.3. The probabilities stem from the fact that there are two possible fixes. Given a DC with more than two atoms, the candidate values contain the conditional probabilities of all possible subsets of atoms. For example, given ϕ_2:$\forall t_1,t_2$:$\neg$(t_1.salary<t_2.salary$\wedge t_1$.age<t_2.age$\wedge t_1$.tax >t_2.tax) which requires that both the salary and the age of the employee define her tax rate, then apart from the aforementioned candidates, we need to include the respective fix of the age field ({3000,0.2,(32 50%,>43 50%)} followed by the pairwise combinations of all three candidate fixes.

Accuracy: DC violations affect result quality since a dirty value might get a candidate fix that satisfies the query. To compute result accuracy, $clean_\sigma$ estimates the theta-join selectivity using the *Estimate_Errors* function of Algorithm 3. The function takes as input the matrix partitions and calculates the overlap of the partition boundaries, that is the number of conflicts between them [92, 67]. For example, consider ranges 3 and 4 of Fig. 4.1, with salary boundaries (3000,4000), (4000,5000) and tax boundaries (0.3,0.4), (0.25,0.5) respectively. The violations lie in the overlap of tax values, that is, (0.25,0.4). Thus, given query answer q_a (line 3), we identify the ranges with which q_a overlaps (e.g., q_1 result overlaps with range 2), and obtain the total estimated errors for these ranges. Assuming inequality conditions, the erroneous partitions that affect the result are the ones with both a *row* and a *column* smaller or larger than the *row/column* of the current range. Otherwise, the erroneous partitions will contain

Algorithm 3: Query-driven cleaning DC violations

```
   input: queries queries, data d, partitions p, rules r, threshold th
1  range_vio = Estimate_Errors(d,p,r)
2  for query in queries do
3      q_a = execute query
4      range = find range of q_a in range_vio
5      errors = {for (i ≠ range) yield range_vio(i)}.sum
6      accuracy = errors/(|q_a|+errors)
7      support = (1 + 2 + ... + √p) − unchecked_p/(1 + 2 + ... + √p)
8      if accuracy > th then
9          full cleaning
10     else
11         partial cleaning
```

either smaller or larger candidate value ranges than the result range. Then, we compute if the estimated accuracy (line 6) exceeds the given threshold (input by the user) and decide to fully or partially clean the data. The range overlap of *Estimate_Errors* function is only applicable over the non-diagonal partitions, since for the diagonal partitions (pattern filled boxes) the ranges are equivalent, thus we also provide the support, that is, the percentage of checked diagonal partitions. The support is defined as the total partitions checked $(1+2+...+\sqrt{p})$, which are the upper/lower diagonal partitions, minus the blocks of the diagonal $(\sqrt{p})$ in the first iteration and becomes smaller depending on the accessed data (line 7). Accuracy also increases while accessing and cleaning more entities.

4.3.3 Cleaning SP results given multiple DCs

In the case of multiple rules, an erroneous cell might trigger violations of many of these rules. Thus, the probability of each fix must combine the probabilities that stem from all the rules affecting the erroneous cell. For the dirty cells that belong to the overlapping attributes of multiple rules, we compute the candidate values in parallel then merge the resulting fixes. We also maintain provenance information for each dirty cell; when many rules exist, we execute them over the original data then merge with the already computed probabilities. Finally, to prune unnecessary error checks, Daisy maintains information about the already checked tuples by each rule.

To merge the probabilities, we compute the overlap of the violating groups. We also use union to merge the candidate values of the overlapping cells, and adjust the probabilities to reflect the union of the sets. Given rules ϕ_1:$Y \rightarrow X$, ϕ_2:$Z \rightarrow X$, we assign $P(X|(Y \cup Z))$ to the X values. Thus, given *zip→state and city→state* and two versions of a tuple with *P(CA|9001)* and *P(CA|LA)*, respectively, one for each rule, the probability will be updated to *P(CA|9001∪LA)*, to match the tuples that have either zip *9001* or city *LA*.

Lemma 4. *In the presence of multiple constraints, the order of computing the candidate values*

of the erroneous cells obeys the commutative property.

Proof. Consider rules ϕ_1 and ϕ_2, which both involve attribute X, and a dirty tuple e which violates both rules. The probabilistic fix of cell $e_x \in X$ of tuple e based on both rules is the same regardless of the order that we check the rules. Consider merging order ϕ_1 followed by ϕ_2. Based on ϕ_1, e_x becomes: e_x=[(a_1,T_1),(a_2,T_2),...,(a_k,T_k)], where $a_1...a_k$ are the candidate values of e_x, and each T_i comprises the conflicting tuples due to which we assign value a_i. For example, for FDs, T_i involves the tuples with the same lhs and different rhs. Then, ϕ_2 produces the corresponding set $e_{x\prime}$. The end result contains the merge of e_x, $e_{x\prime}$, which involves pairs (a_i, T_{mi}), where T_{m_i} is the union of T_i and $T_{i\prime}$. Thus, since the union is commutative, the result is independent of the rule order. □

4.3.4 Cleaning Join results

In the case of join queries, the cleaning operator needs to examine how the existence of errors in each individual table affects the query result. The operator is defined as follows.

Definition 3. $clean_{\bowtie}$ *is a cleaning operator which cleans a join result.* $clean_{\bowtie}$ *(a) extracts the qualifying parts of the join tables, (b) cleans each part and updates each relation separately, (c) updates the result, and (d) re-checks for errors.*

Consider a join between R and S. To clean the dirty result, $clean_{\bowtie}$ extracts and cleans the corresponding qualifying parts of R and S. To extract the qualifying parts of R and S, $clean_{\bowtie}$ keeps provenance information [107] which allows to obtain the entities of each table from the join result, as well as update the join result after cleaning the tables. Thus, using lineage, after cleaning both tables, $clean_{\bowtie}$ recomputes the join to check whether the extra tuples of each relation produce new pairs. In the case where the cleaning task transforms the join key into a probabilistic attribute, the join becomes a probabilistic join. In the following, we show that the updated join accesses the already clean tuples.

Lemma 5. *The updated join result stemming from the cleaned qualifying table parts, requires no extra violation checks.*

Proof. To prove the correctness of the result, we examine the possible correlation cases between the query and the rules. We assume that the result of any erroneous underlying operator in the plan has been cleaned. The possible scenarios depend on whether the join key appears in a constraint. When the join key is clean, the new tuples that relaxation produces will not qualify the join because they will contain a non-qualifying join key. If the join key attribute appears in a rule, then consider a join of R and S which both involve errors on the join key. Then, $clean_{\bowtie}$ might add new tuples to both relations. However, the extra tuples of R will match with tuples of S which already exist in the result since they have to belong to the intersection of the join keys of R and S. Thus, no extra violations will exist. The same case holds for the relaxed result of S. □

	Zip	City
t1	9001	Los Angeles
t2	9001	San Francisco
t3	10001	San Francisco

(a) Cities dataset.

Zip	Name	Phone
9001	Peter	23456
10001	Mary	12345
10002	Jon	12345

(b) Employee dataset.

Zip	Name
9001	Peter

(c) Dirty version of the join result.

Zip
9001
9001, 50% 10001, 50%

(d) Relaxed result of Select Operator over Cities.

C.Zip	E.Zip	Name
9001	9001	Peter
9001, 50% 10001, 50%	9001	Peter
9001, 50% 10001, 50%	10001, 50% 10002, 50%	Mary
9001, 50% 10001, 50%	10001, 50% 10002, 50%	Jon

(e) Clean join result.

Table 4.4 – Join operation over two tables that involve violations on the join key.

Example 6. *Consider tables $Cities$ (C) and $Employee$ (E) shown in Tables 4.4a, 4.4b, rules ϕ_1:Zip→ City, ϕ_2: Phone→Zip, and a query requesting the name, and zip code from both Cities and Employee, for the city of "Los Angeles":*

The dirty version of the result is shown in Table 4.4c. Similarly to Example 2, $clean_\sigma$ enhances the result with tuple t_2 as it belongs to the set *{City≠Los Angeles|Zip = 9001}*. Then, after repairing the detected errors of ϕ_1, tuple t_2 of relation C has candidate values for the Zip $\{9001\ 50\%, 100001\ 50\%\}$. The result of $clean_\sigma$ is shown in Table 4.4d. Then, the evaluation of the join matches the filtered set of C with all tuples of E and $clean_{\bowtie}$ triggers the violation between tuples t_2, t_3 of E. Thus, $clean_{\bowtie}$ fixes the corresponding part and updates the result. The final version of the result is shown in Table 4.4e.

Limitations: Daisy depends on the rules that users provide as input. Therefore, if there is limited knowledge of the rules, or there exists partial dependency among the attributes, then the resulting candidate fixes will be inaccurate. For example, a common approach to infer candidate repairs is by using Bayesian analysis [103] in which the probability of a candidate fix depends on the probability that the dirty tuple shares the same attribute values with the tuple that contains the candidate fix. Such fixes are not covered by Daisy. However, still in that case Daisy can be used in the exploratory process to determine possible worlds based on the existing rules and if the user is not satisfied with the fixes she can execute a full repairing algorithm.

4.4 Cleaning-aware Query Optimization & Planning

In this section, we present how we inject cleaning operators at the logical level, and show a set of optimizations that enable an optimal placement of cleaning operators in the query plan. We support queries with Select, Join and Group-by clauses. The template of the supported queries is the following:

```
SELECT <SELECTLIST>
FROM <table name> [,(<table name>)]
[WHERE <col><op><val> [(AND/OR <col><op><val>)]]
[GROUP BY CLAUSE]
```

[] and () indicate optional and repeated elements. op takes values $=, \neq, <, \leq, >, \geq$. In the case of joins, we assume equi-joins. We focus on flat queries to stress the overhead of the cleaning operators over the corresponding query operators.

4.4.1 Cleaning operators in the query plan

The logical planner detects the query operator attributes that appear in a rule and injects the appropriate cleaning operators in the query plan. The planner pushes cleaning operators down, closer to the data, to avoid propagating errors in the plan. Deferring the execution of a cleaning task causes (a) redundant cleaning to detect errors that have propagated from the underlying query operators, and/or (b) recomputing the underlying query operators that are affected by the errors. For example, consider relations R and S, and a query with a select condition over attribute $R.a$ that participates in a rule, followed by a join $R.a \bowtie S.a$ where $S.a$ participates in a rule of S. Executing cleaning after the join might alter the qualifying part of R by adding an extra probabilistic tuple. If the extra tuple matches with an unseen tuple of S, it will update the join result. Then, the operator needs to re-check for errors over the extra accessed tuples of S, which induces a redundant overhead for query execution. Hence, placing cleaning operators early in the plan avoids extra effort to fix propagated errors.

Fig. 4.3 shows an example query plan with a $clean_{\bowtie}$ operator. Given two rules involving the zip code of R and S, the planner detects the overlap of the join operator with the rules, and injects the $clean_{\bowtie}$ operator. To avoid recomputing the join, $clean_{\bowtie}$ sends only the new tuples of each relation to the join operator. Thus, the second join corresponds to an incremental join, which updates the already computed result. The final result consists of the union of the join outputs.

4.4.2 Cost-based optimization

In the following, we analyze the offline and incremental cleaning cost, and propose a cost model that decides on the optimal placement of cleaning operators in the query plan.

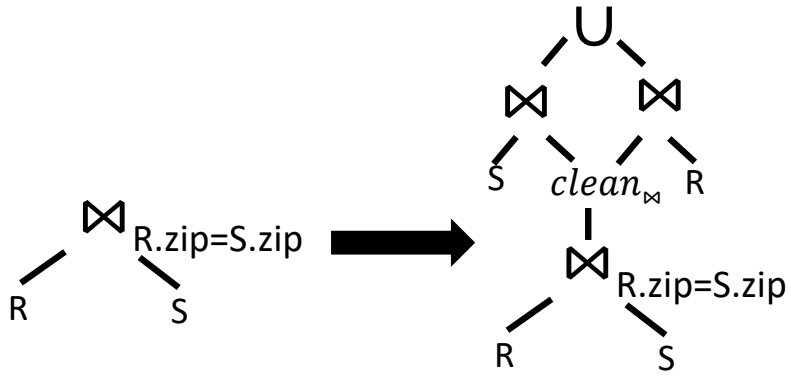


Figure 4.3 – Before and after injecting $clean_{\bowtie}$ inside a plan with a join over a potentially erroneous attribute.

Traditional cleaning cost

The cost of cleaning DCs is divided into the cost of a) error detection, b) data repairing, and, optionally, c) updating the dataset with the correct values. For FDs, error detection groups data based on the *lhs* of the rule. The complexity of grouping assuming a hash-based algorithm over a dataset of size n is $O(n)$ [99]. Similarly, for DCs, the cost is $O(n^2)$ since a cartesian product is required, but is reduced to $(1+2+..+n)$ since the upper diagonal matrix is checked to avoid re-checking symmetric pairs. Data repairing performs multiple scans to compute the candidate values for each error; given ϵ errors, the cost is $O(\epsilon\ n)$. Finally, the update cost is equivalent to an outer join between the dataset and the fixed values; the cost is $O(n+\epsilon)$. The overall cost is: $O(n)/O(n^2) + O(\epsilon\ n) + O(n+\epsilon)$, and can be repeated multiple times if many iterations are needed [27].

Incremental cleaning cost

We present the incremental cleaning cost by taking into consideration the type of query.

SP queries: The cost is equivalent to the cost of computing the correlated tuples plus the traditional cleaning cost. The cost e_i of computing the set of correlated tuples $E(Q)$ is $O(u)$, where u represents the unknown tuples. Given a dataset with n tuples, u is equal to n in the first query, but becomes smaller after each query. Specifically, in the ith query, the cost is $n - \sum_{i=1}^{q-1} q_i$, where q_i is the size of the result of query i.

Error detection and data repairing are applied over the result set $A(Q)$ enhanced with the extra tuples $E(Q)$. Thus, the cost of error detection is $O(q_i+e_i)$ for FDs. For DCs, the incremental cost in the ith query with result size q_i is $n\,q_i/p$. In the worst-case, the incremental cost is the arithmetic progression (we omit the division by p for simplicity):

$$q_1(n-0) + q_2(n-q_1) + ... + q_n(n - \sum_{j=1}^{j=i-1} q_j) = n(q_1 + ... + q_n - \sum_{i=2}^{i=n} q_i \sum_{j=1}^{j=i-1} q_j$$
$$= n \sum_{i=1}^{i=n} q_i - \sum_{i=2}^{n} q_i \sum_{j=1}^{j=i-1} q_j \overset{(\sum_{i=1}^{i=n} q_i \leq n)}{\leq} n^2 - \sum_{i=2}^{n} q_i \sum_{j=1}^{j=i-1} q_j$$

The worst-case is when $\sum_{i=2}^{n} q_i \sum_{j=1}^{j=i-1} q_j$ is minimized to 0, which occurs when one query accessing the whole dataset is executed, thus the cost is equivalent to the offline cost.

For simplicity, we denote the error detection cost as d_i for incremental and df_i for full cleaning. Then, given ϵ_i violated entities, where $\epsilon_i \leq (q_i + e_i) << n$, the data repairing cost is $O(\epsilon_i\ (q_i + e_i))$, since it checks for each error the enhanced tuples instead of checking the whole dataset.

The last step involves updating the original dataset with the fixed tuples stemming from cleaning the query result. The update performs a left-outer-join between the dataset and the clean result. Since the clean result contains probabilistic values, the update depends on the number of candidate values that the erroneous value might take. More specifically, assuming a partially probabilistic dataset at query i with $\sum_{j=1}^{j=i-1} \epsilon_{ij}$ probabilistic values, the update cost is: $n - \sum_{j=1}^{j=i-1} \epsilon_{ij} + \sum_{j=1}^{j=i-1} \epsilon_{ij}\, p + \epsilon_i\, p$, where p is the size of each value. In total, the incremental cleaning cost is:

$$n - \sum_{i=1}^{q-1} q_i + d_i + \epsilon_i \cdot (q_i + e_i) + n - \sum_{j=1}^{j=i-1} \epsilon_{ij} + \sum_{j=1}^{j=i-1} \epsilon_{ij}\, p + \epsilon_i\, p$$

In the case of multiple rules, the cost differs in the error detection part, since it requires one iteration per rule. The computation of the probabilities for the erroneous entities is equivalent to the single rule case because it operates over the detected errors, regardless of the number of violated rules.

Join queries: The aforementioned cost represents the `query and clean` cost of each individual table that participates in a join. However, a join involves the additional cost of updating the join result. Therefore, we need to measure the maximum number of iterations. We apply the cost formula for each dataset that participates in the join, and then we add the incremental join cost which takes place between the extra tuples e_i of one relation with the set of tuples n of the other relation: $(n + e_i)$. We use the formula separately in each dataset, because each dataset has different characteristics, that is different number of violations, different level of correlation among the entities, and finally the query has a different selectivity in each dataset.

Incremental cleaning versus Full cleaning

The decision between incremental or full cleaning depends on whether the overhead induced by the cleaning task in each query is smaller in total than applying the full cleaning followed by the execution of the queries. To estimate the costs, we employ a cost model that decides on the optimal strategy.

Cleaning at query time without considering the relaxation and the update cost, is more efficient overall than executing them over the whole dataset. Consider an unknown query workload consisting of q queries. In the case of FDs, the cost $\sum_{i=1}^{q} \epsilon_i (q_i + e_i) \leq \epsilon\, n$ since, q_i and e_i are complementary, thus, their total number of tuples is smaller than the total dataset. In addition, for DCs, the incremental error detection cost is smaller than the cartesian product. However, the cost of enhancing the query result and updating the dataset after each query might exceed the full cleaning cost. Thus, we decide on cleaning the query result or the remaining dirty part of the dataset based on the following inequality. In the offline cost we also add the query execution cost, which is $q\, n$:

$$\sum_{i=1}^{i=q} (n - \sum_{j=1}^{j=i-1} q_j + d_i + \epsilon_i (q_i + e_i) + n - \sum_{j=1}^{j=i-1} \epsilon_{ij} + \sum_{j=1}^{j=i-1} \epsilon_{ij}\, p + \epsilon_i\, p) \leq qn + df_i + \epsilon\, n + n + \epsilon\, p$$

The inequality can be simplified to the following one:

$$q\, n - \sum_{i=1}^{i=q} \sum_{j=1}^{j=i-1} q_j + d_i + \sum_{i=1}^{i=q} \epsilon_i\, q_i + \sum_{i=1}^{i=q} \epsilon_i\, e_i - \sum_{i=1}^{i=q} \sum_{j=1}^{j=i-1} e_{ij} + p \sum_{i=1}^{i=q} \sum_{j=1}^{j=i-1} e_{ij} \leq df_i + \epsilon\, n + n$$

For example, when $q = 1$, and $\sum_{i=1}^{n} q_i = n$, then $q_1 = n$, $e_1 = 0$ since the query accesses the whole dataset, therefore there are no extra tuples. Thus, the cost corresponds to the full cleaning case and the inequality becomes:

$$n + n + \epsilon\, n + 0 - 0 + 0 \leq \epsilon\, n + 2n \Longleftrightarrow \epsilon\, n \leq \epsilon\, n$$

We observe from the inequality that in the case of general DCs, since p increases when the selectivity is high, then Algorithm 3 will also decide to examine the whole cartesian product

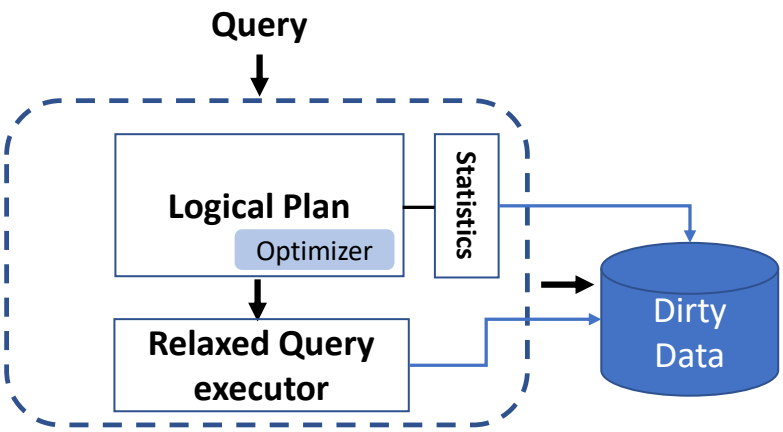


Figure 4.4 – The architecture of Daisy.

due to predicting low accuracy. For FDs, we decide on the cleaning strategy while executing the queries based on the inequality. We estimate the number of erroneous values ϵ, as well as the number of candidate values p using statistics. To approximate ϵ and p, we precompute the group by based on the lhs and the rhs of the FD rules respectively.

4.5 A system for query-driven data cleaning

Fig. 4.4 shows the architecture of Daisy, that is a query-driven cleaning engine over Spark. Given a query and a dirty dataset, Daisy uses two processing levels to provide correct results.

In the first level, Daisy maps the query to a logical plan that comprises both query and cleaning operators. To optimally place each operator, the logical plan takes into consideration the type of query and the constraints. We implement the cleaning-aware logical plan by injecting cleaning operators before/after the corresponding filter and join operators at the RDD level of Spark. Daisy extracts the attributes of the query operators and checks if they overlap with the provided constraints. To apply the cost-based optimizations, Daisy collects statistics by pre-computing the size of the erroneous groups. Then, when checking the condition of each query, it evaluates whether the inequality of Section 4.4.2 holds. Hence, at the logical level Daisy decides whether to place the cleaning operator before or after the query operator.

Finally, Daisy executes the logical plan by cleaning the result of each query operator that is affected by the rules. We implement $clean_{\sigma}$ and $clean_{\bowtie}$ as extra operators inside Spark RDD. The operators take as input the query result, relax it, and detect for violations. Then, given the detected violations, Daisy transforms the query result into a probabilistic result by replacing each erroneous value with the set of values that represent candidate fixes. Daisy also accompanies each candidate value with the corresponding probability of being a fix of the erroneous cell. After cleaning each query result, the system isolates the changes made to the erroneous tuples and accordingly updates the original dataset. By applying the changes after each query, Daisy gradually cleans the dataset.

4.6 Experimental Evaluation

The experiments examine the benefits stemming from the optimizations that Daisy allows, and show how Daisy performs compared to the state-of-the-art offline cleaning approach.

Experimental Setup. All experiments run on a 7-node cluster equipped with 2×Intel(R) Xeon(R) Gold 5118 CPU (12 cores per socket @ 2.30GHz), 64KB of L1 cache and 1024KB of L2 cache per core, 16MB of L3 cache shared, and 376GB of RAM. On top of the cluster runs Spark 2.2.0 with 7 workers, 14 executors, each using 4 cores and 150GB of memory.

We compare a single-node execution of Daisy with Holoclean [103] as it is, to our knowledge, the only currently available probabilistic system for repairing integrity-constraint violations. In the absence of a scale-out probabilistic cleaning system, we compare Daisy with our own offline implementation over Spark; it combines the optimizations of the state-of-the-art error detection and probabilistic repairing systems. Our offline implementation is an optimized implementation that detects FD and DC errors, and it provides probabilistic repairs. Error detection follows the optimizations of BigDansing [66] for FDs; it applies a group-by, instead of an expensive self-join. DC error detection efficiently partitions the cartesian product using the optimized theta-join approach [92]. Directly comparing Daisy with BigDansing would be unfair to BigDansing because BigDansing applies inference in order to compute the correct value, whereas Daisy computes probabilistic repairs. For data repair, to restrict the domain of candidate values for each erroneous cell, we employ an alternative to Holoclean's pruning optimization [103]; we exploit the co-occurrences of the attribute values of the erroneous tuple with the attribute values of other tuples. Hence, similarly to Daisy, the domain of the erroneous rhs of tuple t correspond to the rhs of the tuples that share the same lhs with t. Similarly for the erroneous lhs.

The workload involves SP, SPJ, and group-by queries in the presence of one or more DCs. We evaluate the workload over a synthetic benchmark and three datasets derived from real-world data entries. Specifically, we use the Star Schema Benchmark (SSB) [93], the hospital dataset [103], the Nestle dataset and a dataset with air quality data [1].

We choose the SSB dataset to test the applicability of Daisy over a benchmark designed for data warehousing applications. We use multiple versions of the *lineorder* table by varying the cardinality of the *orderkey* and *suppkey* attributes; we construct different versions by varying the number of distinct *orderkeys* from 5K to 100K, and the number of distinct *suppkeys* from 100 to 10K. To measure the worst-case scenario, we add errors to all orderkeys by randomly editing 10% of the suppliers that correspond to each *orderkey*. Our error generation is similar to BART [15] with the difference that we also add errors using uniform distribution to evenly distribute the errors across the dataset, thereby affecting all queries. The errors that we inject are detectable by the constraints that we evaluate. The size of *lineorder* table is 60MB in the original version, and ranges from 110MB to 2.6GB in the probabilistic version. To evaluate cases with fewer violations, we construct datasets with 20%, 40%, 60% and 80% of erroneous orderkeys. The size of the probabilistic version of those datasets is 250MB, 560MB, 1.3GB, and

1.8GB.

The hospital dataset [103] comprises information about US hospitals. It contains 19 attributes, and is 5% erroneous. We use two versions with 1K and 100K entries, and sizes 300KB, 25MB respectively. The probabilistic versions have size 360KB and 26MB respectively. We use hospital to evaluate accuracy since its clean version exists.

The Nestle dataset includes information about food and drink products. Each product contains 19 attributes and involves dirty categories for product materials. We scale up the dataset by randomly adding duplicate entities from the domain of each attribute. We also add extra errors by randomly editing 10% of the *category* attribute values that correspond to each *material*. We use a 20MB and a 200MB version which contain 95% of conflicting entities. The size of the datasets in the probabilistic version is 40MB and 500MB respectively.

The historical air quality dataset [123] contains air quality measurements for the U.S. counties. We use a subset of the hourly measurements in which we add errors to the FD ϕ:*county_code, state_code*→*county_name*. We edit 10% of the *county_names* that correspond to a *county_code, state_code*. We add the errors to the non-frequent *county_code, state_code* pairs. We use two versions with 0.001% and 0.003% errors respectively, which produce 30% and 97% violations respectively, the size of which is 2GB in the original version and 3.1GB and 4GB in the probabilistic versions.

We measure response time and accuracy (when applicable). Response time is the time to respond to the query, perform the cleaning task by providing probabilistic fixes, and update the dataset. For accuracy, we measure *precision* (correct updates/total updates) and *recall* (correct updates/total errors).

4.6.1 SP queries response time

This section shows how Daisy performs compared to offline cleaning given a workload of SP queries. We measure the cost of both approaches given a) a FD, b) two overlapping FDs, and c) a DC. We evaluate all cases over SSB, by executing queries requesting information for a specific supplier/order, or for suppliers/orders in a given range. In all FD experiments, Daisy outputs the same results with the offline approach.

Single FD with varying selectivity of rule attributes. We examine how the *orderkey* and *suppkey* selectivity affects the response time of the cleaning task. We use three versions of lineorder with 5K, 10K, and 100K distinct *orderkey* values respectively, and three versions with 100, 1K, and 10K distinct *suppkey* values respectively. We use these selectivities since they involve extreme response time cases depending on the query. We clean violations of rule ϕ:*orderkey*→*suppkey*. We consider the worst-case scenario where each *orderkey* participates in a violation. We execute 50 non-overlapping queries, each with selectivity 2%. The workload accesses the whole dataset.

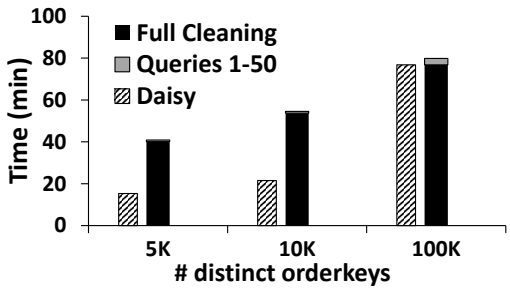


Figure 4.5 – Cost when varying orderkey selectivity.

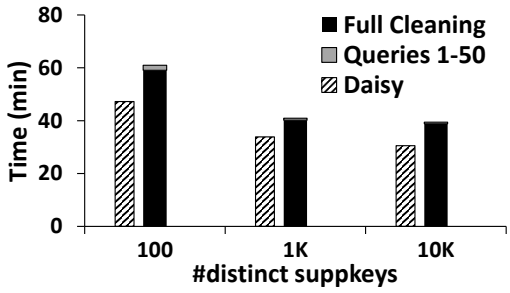


Figure 4.6 – SP Cost when varying suppkey selectivity.

Fig. 4.5 shows the response time of Daisy and full cleaning when varying the *orderkey* selectivity. To maintain a fixed query selectivity, queries contain range filters over the rhs of ϕ. We observe that as the selectivity increases, the response time of both approaches increases. However, on average, Daisy is $\sim 2\times$ faster than the offline approach. The difference is due to the fact that when combining cleaning with querying, query result relaxation restricts the number of comparisons required to repair the erroneous tuples by computing the correlated tuples. On the other hand, the offline approach traverses the dataset for each erroneous value, to compute the candidate values. We also observe that as the selectivity increases, the difference between the two approaches decreases because each erroneous cell ends up having more candidate values thereby increasing the value p of the inequality of Section 4.4.2.

Fig. 4.6 shows the response time of Daisy and offline cleaning when varying *suppkey* selectivity. To maintain a fixed query selectivity, queries contain range filters over the lhs of ϕ. Daisy is faster despite the transitive closure it requires to detect the correlated values. The difference is due to the fact that when combining cleaning with querying, query result relaxation restricts the number of comparisons required to repair the errors by computing the correlated tuples. On the other hand, the offline approach traverses the dataset for each erroneous value to compute the candidates. When *suppkey* selectivity is smaller, the cost becomes higher since each erroneous *suppkey* might match with multiple *orderkeys*, thereby increasing the number of candidate values.

Fig. 4.7 evaluates the scenario in which applying cleaning offline outperforms incremental cleaning (Daisy without the cost model). We execute 90 queries over the lineorder version with 100K distinct *orderkeys*. The queries are non-overlapping, they involve equality and range conditions, and have random selectivities. Cleaning the whole dataset is more efficient in this case because the suppkey selectivity is low compared to the orderkey, thus each suppkey appears with multiple orderkeys throughout the dataset. Thus, a violating suppkey takes multiple candidate values, thereby increasing the update cost shown in the inequality of Section 4.4.2. Still, we observe that overall, Daisy outperforms both the incremental as well as the offline cleaning. Daisy initially applies data cleaning incrementally, and then, by evaluating the total cost after each query, switches strategy and applies the cleaning task over the rest of the dataset. The total cost is lower than the offline approach because cleaning is applied over

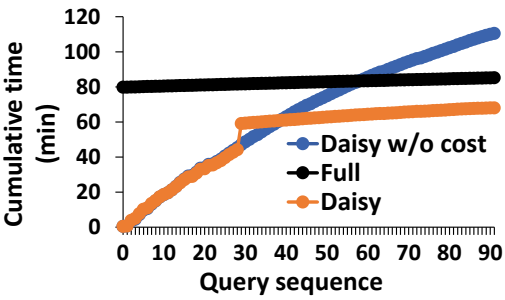


Figure 4.7 – Switching from incremental to full cleaning.

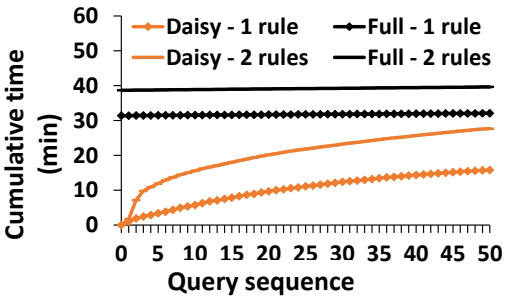


Figure 4.8 – Cost when increasing number of rules.

the remaining dirty part of the dataset.

Single rule vs. Multiple rules. In this experiment, we measure the response time in the presence of rules with overlapping attributes. We construct the dataset by joining lineorder with suppliers. The end result is a 67MB dataset in its raw form, and 2.8GB in its probabilistic form. We evaluate rules ϕ*:orderkey*$\rightarrow$*suppkey* and ψ*:address*$\rightarrow$*suppkey*; the *address* appears after joining the tables. The workload consists of 50 non-overlapping queries which access the whole dataset.

Fig. 4.8 shows the response time in the case where we examine only rule ϕ compared to examining both ϕ and ψ. We observe that in both Daisy and the offline approach, response time increases when we clean errors of both rules instead of one, due to the extra work required for ψ. When Daisy executes the queries, it identifies the corresponding correlated tuples for both rules. Then, Daisy fixes the errors based on the correlated tuples. Initially, the difference between one and two rules is $\sim 3.5x$ but then drops to $\sim 1.5x$ as we clean more data. On the other hand, offline cleaning separately fixes the errors of the *address* and *orderkey* since there might be different tuples involved in the violation of ϕ than those involved in ψ. Thus, offline cleaning needs more traversals over the data.

Increasing number of violations. In this experiment we evaluate Daisy as we vary the number of violations. Specifically, we vary the erroneous orderkeys from 20% to 80%. We use the same query workload consisting of 50 non-overlapping SP queries with selectivity 2%.

Fig. 4.9 shows that in all cases Daisy outperforms the offline approach regardless of the number of erroneous entities. Daisy is faster due to the statistics that it precomputes to prune unnecessary checks. The statistics comprise the orderkeys that participate in a violation; it precomputes a group by based on the orderkey and calculates the size of each group. Then, at query time, when it accesses a specific orderkey, it checks whether it belongs to a dirty group. Thus, Daisy avoids detecting violations when the entity does not belong to the list of dirty values. We also observe that as errors increase, the difference between the two approaches is more significant. The difference stems from the fact that in the case of full cleaning, the number of iterations over the dataset is proportional to the number of detected erroneous

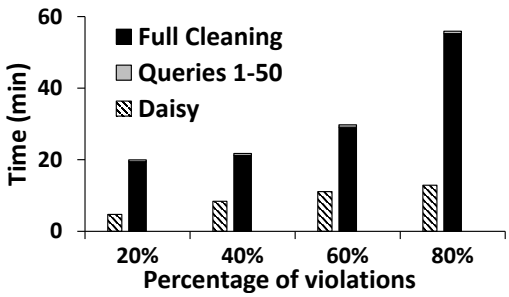


Figure 4.9 – Cost with increasing number of violations.

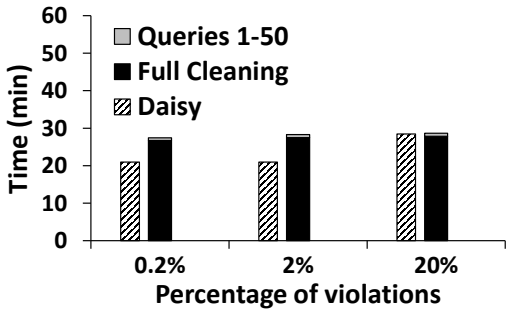


Figure 4.10 – Cost for DCs with inequality conditions.

groups in order to compute the probabilities of each candidate value. On the other hand, depending on the values that the query accesses, Daisy traverses the data once and brings the correlated tuples that correspond to multiple erroneous groups at the same time. Thus, as we increase the number of erroneous groups, offline cleaning performs more traversals, and thus becomes slower.

Denial constraints. In this experiment, we evaluate the cost, given rules with inequality predicates. We consider rule $\forall t_1, t_2 \neg$ *(*t_1*.extended_price<*t_2*.extended_price&*t_1*.discount>*t_2*.discount)*. We check the rule over the lineorder table in which we inject errors by editing the *discount* value of 10% of entries. We simulate real-world scenarios that, unlike high selectivity inequality joins, contain a few dirty values that cause inconsistencies. We construct three versions with 0.2%, 2%, and 20% violations, by modifying the errors that the dirty values induce. We execute 60 SP non-overlapping range queries that access the whole dataset.

Fig. 4.10 shows the response time of both Daisy and the optimized offline approach. In the 0.2% and 2% versions, Daisy is 1.3× faster, as it prunes both the partitions and the subset of the partitions that must be checked. The result is 99% and 80% accurate, respectively, compared to the offline case. In the 20% case, Daisy predicts 23% accuracy by using the statistics, hence it decides to clean the whole dataset and is 100% accurate and has the same response compared to the offline case. Specifically, in the 20% case, the dirty values are spread across different partitions and contain outlier values that affect the result. Therefore, this case justifies the need for checking the whole matrix to provide an accurate result.

4.6.2 SPJ queries response time

This section demonstrates how Daisy performs when Join queries appear in the workload. We execute 50 join queries over lineorder and suppliers. The lineorder table violates rule ϕ*:orderkey*$\rightarrow$*suppkey*, and the suppliers violates rule ψ*:address*$\rightarrow$*suppkey*. The queries contain a filter on lineorder, and then join it with suppliers. The workload accesses the whole lineorder dataset.

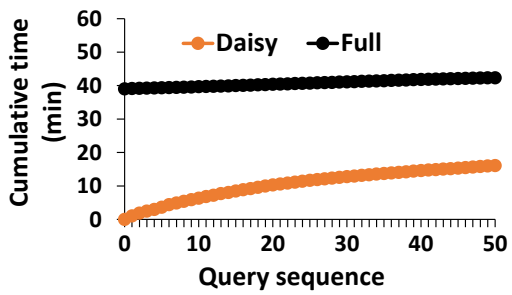


Figure 4.11 – Cost for join queries.

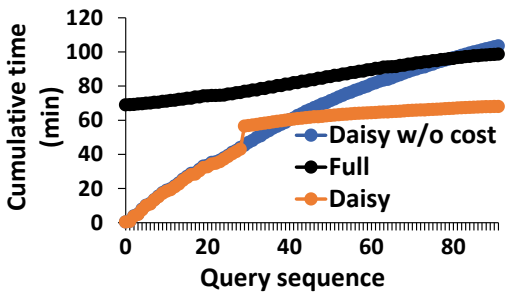


Figure 4.12 – Cost for mixed workload.

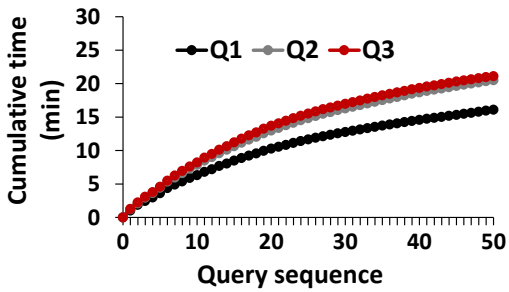


Figure 4.13 – Cost for complex queries of SSB workload.

Fig. 4.11 shows the response time of SPJ queries using Daisy and the offline approach respectively. Daisy outperforms full cleaning for two reasons: First, similarly to the SP queries case, Daisy benefits from computing the set of correlated tuples, thereby restricting the number of comparisons. Second, Daisy benefits from incrementally updating the join result when extra tuples are added. On the other hand, offline cleaning performs a probabilistic join which is expensive.

Fig. 4.12 shows the time taken to execute a workload with both SP and SPJ queries. The lineorder table violates rule ϕ:*orderkey*$\rightarrow$*suppkey*, and the suppliers violates rule ψ:*address* $\rightarrow$*suppkey*. We use the scenario of Fig. 4.7, where we execute 90 queries over the 100K version of *lineorder*, and the suppkeys contain 500 distinct values. Both the SP and join queries are non-overlapping, involve equality and range conditions and have random selectivities. We observe that Daisy predicts that it is more efficient to clean the full dataset after 30 queries, and thus by penalizing some queries, overall it is faster than both incremental and full cleaning.

Fig. 4.13 compares the response time of three query workloads from the SSB family to evaluate how Daisy behaves with more complex queries. We use the same setup with Fig. 4.11. *Q1* is a join between *lineorder* and *suppliers* and contains a range filter on the *suppkey*. *Q2* additionally joins the result of *Q1* with *part* and *date* tables and groups by year and brand. *Q3* contains a fourth join with *customer*. All queries project the keys of the involved tables thus the probabilistic orderkey/suppkey attributes as well. We observe that regardless of the query complexity, since Daisy pushes down the cleaning operator, cleaning affects only the join

	ϕ_1			ϕ_1+ϕ_2			ϕ_1+ϕ_2+ϕ_3		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Holoclean	1	0.55	0.71	0.98	0.95	0.96	0.98	0.92	0.95
DaisyH	0.97	0.52	0.68	1	0.98	0.99	1	0.98	0.99
DaisyP	0.41	0.51	0.45	1	0.97	0.98	1	0.98	0.99

Table 4.5 – Accuracy

between *lineorder* and *suppliers*. The breakdown of the cost of the overall plan showed that the time difference between *Q1* and *Q2,Q3* stems from the fact that in *Q2,Q3* the initial join projects the extra attributes required for the following joins. Thus, cleaning is more expensive since Spark requires outer joins to split and stitch back the clean and dirty part of the query result.

4.6.3 Real-world scenarios

In this set of experiments, we compare Daisy against Holoclean, and we also measure the cost of executing a realistic exploratory analysis scenario. In all experiments, Daisy outputs the same results with the scale-out cleaning approach.

Hospital: We evaluate the efficiency and accuracy of Daisy and Holoclean. We use rules ϕ_1:$\forall t_1,t_2$:$\neg(t_1.zip=t_2.zip \wedge t_1.city \neq t_2.city)$, ϕ_2:$\forall t_1,t_2$:$\neg(t_1.hospitalName=t_2.hospitalName \wedge t_1.zip \neq t_2.zip)$, ϕ_3:$\forall t_1,t_2$:$\neg(t_1.phone=t_2.phone \wedge t_1.zip \neq t_2.zip)$. To obtain a fair comparison, we execute Daisy on a single node, and when measuring response time, we disable the inference of Holoclean; we obtain only the candidate values for each cell. For accuracy, we apply Holoclean's inference using Daisy's domain generation (DaisyH), and we compare it with the original Holoclean. To integrate Daisy with Holoclean, we populate the *cell_domain* table that Holoclean uses with the candidate values that Daisy computes. We also report Daisy's accuracy when selecting the most probable value (DaisyP). For accuracy, we use the 1K version for which master data exists. For efficiency, we use version 100K.

Table 4.5 shows the precision, recall, and F1-measure for Daisy and Holoclean. Daisy executes a workload of 4 SP queries that access the whole dataset. Each tuple is accessed only once and is cleaned at query time. For Holoclean we clean errors offline and measure the accuracy of the corresponding attributes. We observe that both systems exhibit comparable accuracy. When not all rules are known, such as in the case of ϕ_1, Holoclean performs better because it generates the domain using quantitative statistics, whereas DaisyH uses the correlations driven by the dependencies. DaisyP performs worse because it blindly selects the most probable value. However, when more rules are known, Daisy is more accurate because Holoclean prunes the domain of each value using a threshold for performance. Hence, using Daisy's optimizations, one can avoid trading accuracy at this level.

Table 4.6 shows the benefit stemming from maintaining provenance information to the original

	ϕ_1	$\phi_1 + \phi_2$	$\phi_1 + \phi_2 + \phi_3$	Total
Daisy (3 executions)	51 sec	49 sec	118 sec	218 sec
Daisy (1 execution)	51 sec	41 sec	40 sec	132 sec
Holoclean	1020 sec	1108 sec	1188 sec	3316 sec

Table 4.6 – Response time when increasing the number of rules. Daisy maintains provenance information and updates the probabilistic data based on the new rule without having to execute the task from scratch.

data and incrementally updating the probabilistic data in the case new rules appear. We measure the total cost by checking ϕ_1, $\phi_1 + \phi_2$, and $\phi_1 + \phi_2 + \phi_3$. We use a scenario where we execute Daisy and Holoclean three times, one for each rule set, and compare it with a single execution of Daisy that incrementally updates the probabilistic data. We evaluate the case where a user queries the whole dataset and executes the cleaning task, thus the cost of Daisy is equivalent to the offline cost. For Holoclean, we measure only the cost of the candidate fixes for each cleaning task. We observe that the single execution of Daisy outperforms the three separate executions since it can merge the probabilistic fixes by inducing only the overhead of merging the fixes.

Nestle exploratory analysis: Data scientists working for Nestle, often need to apply analysis to discover information about different coffee products. We simulate this scenario and execute a query workload of 37 SP queries in which the analyst requests the details of a given coffee product through the $Category$ attribute. The dataset contains violations of the FD $Material \rightarrow Category$. $Material$ represents the material out of which each product is made; in the case of coffee products it represents the type of beans. $Category$ is the type of product.

Table 4.7 shows the response time of the analysis over the two versions of the dataset. In both cases, the queries access 40% of the dataset. We observe that in the smaller dataset (20MB), the difference in the response time stems only from the fact that the analysis accesses 40% of the dataset. However, when the dataset becomes bigger, the difference is more significant. Daisy is faster because the selectivity of the $Category$ attribute is very small, and thus since it appears with multiple erroneous $Material$ values, the full cleaning approach ends up iterating through the dataset multiple times.

Air quality exploratory analysis: This scenario is similar to the analysis that data scientists perform in Kaggle [123], where they observe how air pollution evolves over the years in the US. Specifically, an analyst checks the CO measurements at specific locations, one location per state, district, or territory. The query workload consists of 52 queries each of which outputs the average CO measurement for a given county grouped by year. Table 4.7 shows that offline cleaning is unable to terminate after a timeout of one day due to having to perform multiple iterations for each erroneous group over a larger dataset in order to clean it.

Summary. The optimizations at the executor level ensure that Daisy scales better than offline

Dataset	Daisy	Offline
Nestle (20MB)	2.9 min	3.97 min
Nestle (200MB)	26.8 min	8.5 hours
Air quality 30%	10.5 min	-
Air quality 97%	49 min	-

Table 4.7 – Response time on realistic scenarios.

approaches by restricting the comparisons to clean the data. Moreover, the logical-level optimizations enable Daisy to configure the optimal placement of cleaning operators, depending on the query workload and the errors.

4.7 Summary

Data scientists usually perform multiple iterations over a dataset in order to understand and prepare it for data analysis. Having to apply each cleaning task over the whole dataset each time is tedious and time-consuming. Having data cleaning decoupled from data analysis also increases human effort as data cleaning highly depends on the data analysis that users need to perform.

Our work introduces Daisy, a system that partially cleans the dataset through exploratory queries. Daisy integrates cleaning operators inside the query plan, and efficiently executes them over dirty data by providing probabilistic answers for the erroneous entities. We evaluate daisy using both synthetic and real workloads and show that it scales better than approaches that fully clean the dataset as an offline process.

5 Scalable Graph Path-Matching

5.1 Introduction

Graphs have been widely used for data analysis, thanks to their inherent capability of representing connections among entities. Due to their generality, graphs are of great practical value in interactive data exploration where users look for interesting patterns in lakes of structured and unstructured data. Several use cases stemming from social network, web search, log trace analysis and many more, require to identify parts of the graph that match specific patterns [109]. However, together with the degrees of freedom in data exploration, subgraph matching brings high computational complexity. Thus, query execution needs to efficiently address complexity to produce timely results.

Given a labelled data graph G, subgraph matching is the problem of finding all matches of a labelled pattern graph Q in G. Labelled graphs are the graphs which have certain properties defined as labels in their vertices and edges. Thus, a match of the labelled graph Q in G is a subgraph G' of G that is isomorphic to Q, i.e., for each vertex in the query pattern, (a) the vertex appears in G' and has the same label, and, (b) for each edge appearing in the query pattern, the edge also appears in G' and has the same label. An example pattern is *person* $\rightarrow_{isLocatedIn}$ *place* which looks for subgraphs consisting of two vertices with label *person* and *place* respectively, and that are connected with the edge *isLocatedIn*. The subgraph matching problem is equivalent to the subgraph isomorphism problem which is known to be NP-complete [30].

To efficiently execute subgraph matching tasks, existing approaches propose optimizations by assuming either a relational or a graph layout. In the relational layout, graphs are represented through property tables [6]. Property tables are relational tables that represent a property of the graph and contain two columns that represent the vertices that are connected through that specific property. By using the relational representation subgraph matching is reduced to executing a multi-way join, therefore existing approaches focus on join optimization for graphs by developing worst-case optimal join algorithms [11]. Approaches that follow the graph layout optimize subgraph matching by limiting checks within minimal candidate regions

[56, 19]. Parallel graph-based solutions introduce extra steps either offline [75] or online [121] to optimally partition the work in order to minimize communication, as well as to balance the load. Graph-based systems that target explicitly the subgraph matching problem take the workload into consideration [84, 45] by introducing offline, workload-aware partitioning. Relational approaches are both inappropriate and inefficient for representing graph data as they require flattening the graph by storing redundant information [64]. On the other hand, subgraph matching approaches that are based on the graph layout assume expensive preprocessing to optimally partition the graph for the given workload.

This thesis targets path queries over labelled graphs, which constitute a class of widely used types of patterns in interactive data exploration applications like fraud detection and social network analysis [12, 84]. In this line of research, G-core [12] proposes a query language for graphs and highlights the importance of path queries by extending property graphs with path property graphs, whereas property paths [5] have been also integrated in SPARQL [4] to allow navigation over RDF documents.

Further, we focus on a scale-up setting, following the recent works which demonstrate that graphs can conveniently fit in the main memory of modern multi-socket, multi-core servers [105, 100]. The scale-up setting is different than the scale-out, as the latter is mainly affected by the degree of partitioning, since remote partition accesses over the network are the main bottleneck. There are two fundamental factors that limit scalability of subgraph matching in scale-up deployments. *First*, the synchronization of work across the threads. Qualifying paths may span across several partitions owned by different worker threads. Thus, some threads will end up accessing each other's partitions while following the same path that matches a pattern before synchronizing and deciding to keep only one resulting path. *Second*, the load of work of each thread. As the partial paths discovered by each thread may largely vary, depending on the properties of the graph, parallel pattern evaluation will lead to load imbalance.

We propose a work-sharing approach that overcomes the bottlenecks of scale-up graph pattern matching by coordinating the threads to avoid redundant work and minimize intermediate results, as well as to balance the pattern matching load. In doing so, we make the following contributions:

- We propose a parallel, online graph path matching algorithm where workers share partial results by incrementally building a radix tree. Through the radix tree, workers progressively discover the requested paths, even if they span across multiple partitions, while avoiding duplicate work in the path evaluation. We also optimize the radix tree construction to minimize intermediate results. Specifically, we (a) progressively exchange information among the workers to prune partial patterns with no valid continuation, and (b) we defer the evaluation of high degree vertices to avoid large intermediate results with no valid continuation.
- We introduce online work-stealing in the optimization of the path evaluation and show

that work-stealing fully compensates for balanced, workload-aware graph partitioning without additional cost.

- We experimentally demonstrate the scalability of our approach by considering different benchmarks, path lengths, and worker thread count.

5.2 Preliminaries

Consider a labelled graph $G(V, E, L)$ where V is a set of vertices, E is a set of edges, and L is a set of labels assigned to vertices and edges. Labels might involve any domain-specific value. Edge labels represent the relationships through which the vertices are connected. A labeled graph is therefore a heterogeneous graph as it contains different types of vertices and edges.

Subgraph Isomorphism of labeled graphs: Given a graph $G_p(V_p, E_p, L_p)$, find a subgraph $G'(V', E', L')$, of G, such that G' is isomorphic to G_p, i.e., there exists a bijective function $h : V' \rightarrow V_p$ such that (i) for each vertex u' in G', the vertex appears in G_p ($u' \in V_p$) and has the same label as u': $L'(u') = L_p(h(u'))$, and, (ii) for each edge $e'(u', v')$, $u', v' \in V'$ appearing in G', the edge also appears in G_p and has the same label: $L'(e'(u', v')) = L_p(e'(h(u'), h(v')))$.

Subgraph Matching: A match of graph G_p in graph G is a subgraph $G' = (V', E', L')$ of G such that G' is isomorphic to G_p. Specifically, the subgraph matching problem looks for all distinct embeddings of G_p in G. An example subgraph is $Person \rightarrow_{isLocatedIn} Place$ which looks for pairs of vertices with type $Person$ and $Place$ respectively, and are connected with the $isLocatedIn$ edge type.

Path Query: A labelled path query [79] comprises a set of labelled vertices which are connected through a set of labelled edges. A labelled path query is of the following form:

$$\pi = v_1 e_1 v_2 e_2 .... e_{n-1} v_n$$

such that each v_i is a vertex in V, and each e_i is an edge in E. The path π is accompanied by a data path which is a sequence of alternating vertex labels and edge labels that correspond to each v_i and e_i respectively. Path queries constitute a common subcategory of subgraph matching queries.

Subgraph isomorphism problem. The subgraph isomorphism problem is an NP-complete decision problem due to reduction from the clique problem. However for the case of fixed patterns, the enumeration of all possible subgraphs requires polynomial time $O(n^l)$ where n is the number of vertices of the graph and l is the number of vertices of the pattern. The basic subgraph isomorphism algorithm is implemented as a backtracking algorithm [77] which finds solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed. Common state-of-the-art algorithms for subgraph isomorphism are the VF2 algorithm [30] and Ullmann's algorithm [124, 125].

5.3 Scaling up path matching

There are two main approaches for subgraph matching: (a) a Depth-First-Search (DFS) [30, 124] or Breadth-First-Search [121] graph traversal over raw graph data, or (b) a BFS traversal represented as a set of joins or self-joins over a relational representation of the graph [126, 53]. A DFS traversal requires traversing the neighboring vertices of a vertex multiple times, once for each path it participates in. Therefore, it induces additional, and unnecessary work. On the other hand, a relational representation materializes the whole graph as a single or more relational tables and employs existing multi-way join optimization approaches [11]. Apart from the additional cost of materializing the graph, relational representation is inefficient for graph data as it requires flattening the hierarchical graph representation. Specifically, either by using a single relational table or multiple property tables to represent relationships, each vertex will be stored multiple times, once for each neighbor, thereby introducing redundant information [64].

Apart from the graph traversal challenges, scaling up the graph representation and traversal for path matching, poses a fundamental challenge in distributing the parallel tasks to independent workers. In general, parallel algorithms follow two approaches to distribute work among threads. One approach needs to first partition the data and then execute data-parallel operations on the individual partitions. Ideally, if there are no dependencies among operations performed in different partitions, we obtain significant speed-up. However, graph partitioning is known to be NP-complete [75] and has been shown to be non-beneficial for path matching when there is no information about the workload [45, 84]. Thus, a parallel algorithm cannot rely solely on a partitioning scheme in order to scale. The second approach repartitions the partial paths at runtime [121] to synchronize work among threads as well as to balance the load. Executing synchronization and load-balancing at runtime poses however a significant overhead.

In this section, we present a two-phase algorithm for graph path matching, which is *adaptive to the workload* by distributing the work among threads at runtime and in different stages. We start by giving an overview of the approach and then we explain each phase in detail.

5.3.1 Overview

We consider a widely used graph representation that uses two data structures: the *vertex list* and the *adjacency list*. The vertex list stores the vertices of the graph along with their labels. The adjacency list stores for each vertex its neighboring vertices. The adjacency list also maintains for each neighboring vertex of a vertex the edge label based on which the vertex is connected to the neighbor.

We assume non-recursive paths where each vertex label appears at most once in the input path. We perform path matching in two phases: the first phase is responsible for clustering the data based on qualifying partial sub-paths, and the second phase is responsible for combining the

partial sub-paths to form paths that satisfy the query path. Both phases minimize intermediate results by integrating a progressive filtering mechanism inside the process of clustering and merging the partial states.

5.3.2 First-phase: Cluster, Filter, and Index

The parallelization of path matching requires the separation of the data to allow the worker threads to execute their tasks in parallel. Contrary to scale-out, in the scale-up context that we study in this thesis, the problem of data shuffling across threads is not applicable. Instead, the main problems are (a) redundant work as a result of sub-optimal thread-to-data assignment, and (b) large intermediate results. In the following, we first explain the aforementioned problems, and then we provide our solution which is based on radix clustering.

Consider an arbitrary graph partitioning scheme which assigns a graph partition to every worker thread. The partition will include a set of vertices and edges. A path matching algorithm will start from a given vertex and explore the neighborhood exhaustively, typically following an approach based either on Breadth-First-Search (BFS) or on Depth-First-Search (DFS). While exploring candidate paths, worker thread i will generally have to visit the partition of worker thread j, unless a perfect partitioning scheme has been used. To the best of our knowledge, there is no such partitioning scheme that fits any exploratory workload without requiring startup cost for repartitioning. However, this also means that worker thread j will also have to visit the partition of i. Therefore, the same path will be considered two times. As the graph becomes denser and the path longer, redundancy increases.

The first phase of our approach is the driver of the algorithm which explores the candidate paths, by clearly separating the search space at every step through a radix clustering scheme. The process is parallelized and it can be applied over any kind of graph partitioning scheme.

Clustering: Radix clustering separates the candidate paths in clusters based on the part of the path that they match. We use a radix identifier to index every cluster, where every bit which is set represents a matching part in the query path. The size of the radix identifier is equivalent to the path size. Specifically, given a query path of size p, a radix cluster identifier has the regular form $\{0^*1^+0^*\}^p$. For example, given an input path $Gp : Person \rightarrow University \rightarrow City \rightarrow Country$, a detected sub-path that consists of a vertex of type $City$ will get the radix 0010. The incentive for using the radix identifier is that it adds semantics to the path matching process, and therefore it allows us to easily trace the sub-paths which are complementary and can be merged to construct larger paths closer to the query path. The radix clustering phase yields two benefits: (i) filtering out the paths which do not match the path as early as possible, and (ii) enabling independent execution of several worker threads over different clusters.

The use of the radix clustering idea in our context depends on the query (workload) and not the database, as happens for instance in the case of radix joins [22]. Accordingly, we draw inspiration from the related literature and we leverage the parallelization properties offered by

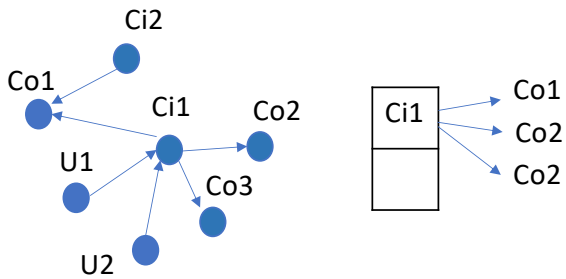


Figure 5.1 – Filtering and indexing for the path $Person \rightarrow University \rightarrow City \rightarrow Country$. Index only the vertices that have a valid continuation, e.g., store $Ci1$ and prune $Ci2$.

radix clustering, *while making our execution algorithm workload-aware.*

Pre-Index Filtering: While constructing the radix clusters, we also prune *non-qualifying vertices*, *non-qualifying neighbors*, as well as *vertices with no viable continuation* with respect to the input path. The filtering takes place in a progressive, transitive fashion, that is apart from obtaining information from the neighboring vertices and edges, the filtering phase obtains information from previous checks in order to further prune vertices that despite being qualifying, they do not lead to a qualifying path. In the following, we present the general filtering approach which resembles existing approaches [56] and then we also present how we further prune non-qualifying vertices using transitivity.

Non-qualifying vertices are the vertices which have a label that does not participate in the path. The *non-qualifying neighbors* of a vertex v are the neighbors that violate the feasibility criterion of satisfying the input path. The feasibility criterion is based on the subgraph isomorphism definition and states that the violation takes place if (a) the neighboring vertex is not a qualifying vertex of the path, or (b) the neighboring vertex qualifies the path, but the edge connecting v and the neighboring vertex does not qualify the path. In that case, the neighboring vertex is pruned. The *vertices with no viable continuation* are the vertices that, depending on their radix index, lack either the previous qualifying neighbor for the path or the next qualifying neighbor. This type of filtering constitutes a look-ahead filtering to avoid considering a vertex that will not lead to a valid path. To estimate the viability of a qualifying vertex v, the filtering step iterates through the neighbors of v, computes their position in the path, and it checks if they qualify for being a continuation of the path involving v based on their edge label with v. We optimize the process of checking whether the vertex v has a possible continuation by limiting the number of checks; in the case where v appears in the position p_v of the path and $p_v > 0$, that is it is an intermediate vertex, then the filtering process checks whether there exists at least one vertex for which the position is $p_n < p_v$ and which satisfies the criteria for being a continuation of the path. Thus, it avoids checking the feasibility criteria for continuing a path for all neighboring vertices.

To extend the look-ahead pruning rule and further limit the vertices considered for the path matching process, we use a transitivity rule that transfers the knowledge of the already non-qualifying vertices to their neighbors. Specifically, consider that the filtering process checks

the filtering criteria by traversing the vertices list. Given that the filtering checks first vertex v followed by vertex u, and that it discovers that vertex v has a no valid continuation as there is no qualifying vertex in the position $p_n < p_v$ of the path. Then it will mark vertex v as non-qualifying. Therefore, while processing vertex u, if u is a neighboring vertex of v and v is the only qualifying neighbor of u, then u will also be pruned and marked as non-qualifying.

The filtering step resembles a materialized property graph [126] with three key differences: (i) it is built online based on a given path, (ii) it is workload-aware as it does not require materializing all possible pairs of vertices that are connected with any relation, and (iii) it prunes vertices that are not candidates for a path. In the example graph shown in Figure 5.1 where labels U, Ci, Co represent universities, cities, and countries respectively, for the vertex with label Ci_1 it is sufficient to visit only the neighboring university vertex U_1 in order to verify whether it has a valid continuation prior to Ci_1. In addition, even though the vertex with label Ci_2 qualifies the path, it will be pruned as it does not have a valid continuation on the one side.

Indexing: We index each and every qualifying vertex v that belongs to a radix cluster by storing it in a hash table. There is a single hash table per cluster, which keeps all the vertices of the cluster. Indexing is pipelined with filtering, and every vertex that passes the filter is indexed immediately. Further, there is no coordination requirement because each worker thread is responsible for a disjoint set of vertices, therefore no two threads will attempt to modify the same entry in the hash table. Therefore, the index data structure is completely lock-free.

Algorithm 4 presents the first phase of the path matching process. Each worker gets as input a query path G_p and a partition of the graph for which it is responsible. The algorithm instantiates an empty path and then executes the filtering phase where it searches for qualifying vertices (lines 1-3). Once it detects a qualifying vertex v, it computes the radix cluster of the vertex. As the cluster is based on the position of the vertex in the path, we compute it based on the mapping of the type of vertex with the corresponding vertex of the subgraph. Thus, given that $vertexType$ is the type of vertex denoted as the label associated with the vertex, the radix_cluster is defined as the cluster of the vertex based on its position (line 6). The result cl_v of radix_cluster will be of the form 2^{pos} if the vertexType of the vertex is the position pos of the path. Afterwards, the algorithm interleaves the filtering and indexing phase. Specifically, it checks whether there exists a valid continuation of a path that contains the vertex v. Thus, it iterates through the neighbors of the vertex and maintains the qualifying neighbors in the set $possible\_continuation\_set$ (lines 8-14). As explained above, the $possible\_continuation\_set$ must contain only vertices which are after v in the path for efficiency. The algorithm only checks whether there exists at least one vertex on the left of v to ensure the viability of v for a path. Finally, it indexes each qualifying vertex v in the hash table of the corresponding radix cluster cl_v of v by also storing each qualifying neighbor (lines 15-19).

For example, consider the graph shown in Figure 5.2 which is partitioned randomly in two

Algorithm 4: First iteration: Filter, Index & Cluster

```
input: Graph G(V_G, E_G), Query pattern G_p,
       Partition partition
1  Path path = {}
2  foreach Vertex v in partition do
3      if path.qualifying(G_p, v) then
4          has_valid_left = false
5          path.append(v)
6          cl_v = radix_cluster(v.vertexType, G_p)
7          possible_continuation_set ={}
8          foreach Vertex nextv in neighbors(v) do
9              if path.qualifying(G_p, nextv) then
10                 cl_nextv = radix_cluster(nextv.vertexType, G_p)
11                 if cl_nextv < cl_v or cl_v == 2^|G_p| then
12                     has_valid_left = true
13                 else
14                     possible_continuation_set.append(nextv)
15         if has_valid_left &
16         !possible_continuation_set.empty then
17             foreach Vertex nextv in possible_continuation_set do
18                 ht[cl_v][v].append(nextv)
19         path.remove(v)
```

partitions shown in blue and red respectively. Consider also the path Gp. The first phase of the algorithm computes three possible sub-paths for the blue partition and one sub-path for the red partition. Each sub-path will keep a pointer to the next qualifying neighbors. For example, $U1$ will keep a pointer to $Ci1$, as it is the next valid vertex given the input path. Each of the qualifying sub-paths are indexed in order to efficiently retrieve them in the Join&Merge phase as explained in the following section.

Figure 5.1 presents the indexing step in more detail. In the case of vertex $Ci1$ as it has three possible qualifying neighbors, we index only the neighbors in order to avoid replicating $Ci1$ for each candidate path.

5.3.3 Second-phase: Join and Merge

The second phase consists of joining and merging the candidate partial paths discovered during the first phase. To merge the partial paths, the second phase incrementally constructs a *radix cluster tree*. Specifically, for a query path G_p of length p, we split the intermediate results into $2p-1$ clusters, which are connected in a perfect binary tree with p leaves. The leaf clusters have only one bit set to 1 and there are no two leaves with the same bit set to 1. Every pair of nodes are joined into a new cluster whose identifier is the result of the bit-wise XOR of the two joined clusters identifiers. Naturally, our algorithm starts from the leaves and

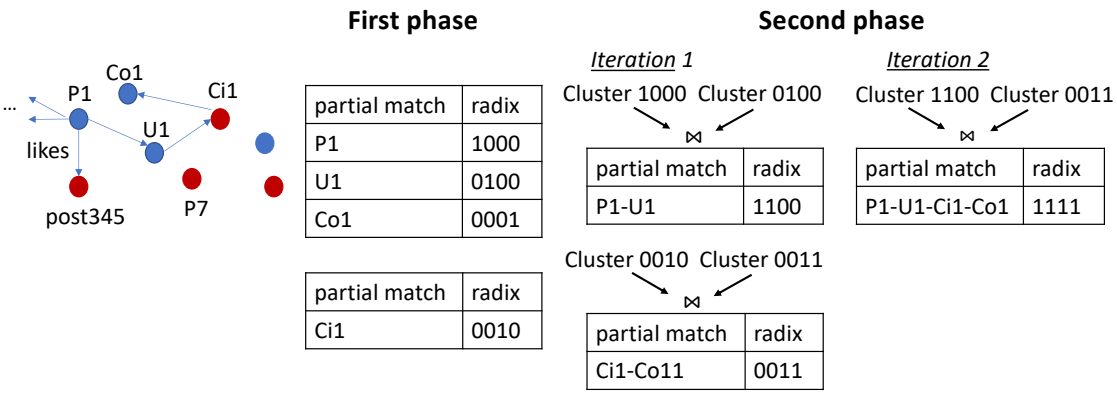


Figure 5.2 – Path evaluation example for the input path
$Person \rightarrow University \rightarrow City \rightarrow Country$

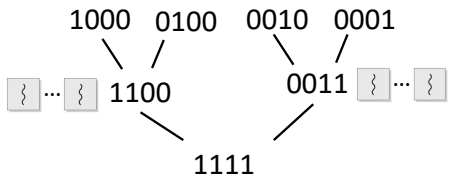


Figure 5.3 – Radix tree

fills all the clusters until the root, which includes the final results. Figure 5.3 depicts a radix cluster tree for a query path of size four, which is progressively constructed from the top to the bottom.

The construction of the tree takes place by incrementally joining and merging partial sub-paths. To speed up the execution, we interleave the join and merge operations. As shown in Figure 5.3, the radix cluster tree resembles a bushy tree because, while it allows for an efficient parallel execution, it also bounds the number of merge operations required.

The radix tree stores in memory each time the current level and the next level in order to perform the join. It garbage collects any previous unnecessary levels. In addition, each in-memory hash table that the radix tree requires stores only the qualifying sub-paths for the given input path, therefore it does not require materializing and keeping any unnecessary data. In the following we present the Join and Merge operations and how we scale up their execution.

Join. The Join operation builds and traverses the tree top-down and joins adjacent clusters at each level. Specifically, at each level, it joins clusters n and $n+1$ with corresponding radix cluster indexes of the regular form $\{0^*1^+0^*\}^p$. The constraint that the adjacent nodes of each level satisfy is that the first 1 bit of node $n+1$ is in the next position of the last 1 bit of node n. Notice that there is an ordering constraint because in order to be able to join cluster i with

Algorithm 5: Join and Merge

```
   input: Radix tree RT, pattern L
1  foreach level k in RT do
2      join_merge_level(RT[k], L)
```

Algorithm 6: Join and Merge for level k

```
   input: Radix tree level RT_k, pattern size p
1  partial_size = 2^k
2  for i ← 0 to p/2^k by 2 do
3      s_start = i * partial_size
4      t_start = (i + 1) * partial_size
5      cl_s = Σ_{s=p−s_start}^{s=p−(s_start+partial_size)} 2^s
6      cl_t = Σ_{t=p−t_start}^{t=p−(t_start+partial_size)} 2^t
7      cl_r = cl_s ⊕ cl_t
8      foreach subpath in ht[cl_s] do
9          foreach next_v in ht[cl_s][subpath.start] do
10             matches=probe(next_v, ht[cl_t])
11             foreach match in matches do
12                 merged_sub_path = merge(subpath, match)
13                 ht[cl_r][subpath.start].append(merged_sub_path)
```

cluster j, all clusters that are above cluster j in the tree must have been joined and merged. For example, the join of clusters 0100 and 0011 must take place after joining 0010 with 0001; otherwise, there will be incomplete results.

Merge. When joining clusters i and j, we also merge the partial paths that they contain and push them to the radix cluster k, where the following condition holds on their identifiers: $k_{id} = i_{id} \oplus j_{id}$. For example, the merge of clusters 0010 with 0001 will be stored in cluster with radix 0011.

The merge operator merges the partial paths without the need to check the feasibility criteria of extending the path of one cluster with the matching sub-path of a neighboring cluster identified by the join operation. We push down the feasibility check in the filtering phase of the first step which filters only neighboring vertices that constitute a valid continuation of a vertex. Therefore, given that we merge partial paths involving consecutive positions in the path, and that the neighbor is a qualifying continuation, this ensures that the merge can take place with no extra check.

Algorithms 5, 6 present the join and merge operation given as input the first level of the radix cluster tree, constructed in the first phase. Algorithm 5 traverses the radix cluster tree level-by-level. At each level it populates the nodes of the next level. Then, as shown in Algorithm 6, at each level, it takes as input the clusters constructed from the previous iteration, and tries to join them. Consider the k-th iteration which joins the clusters of level k of the tree. Let us

assume for simplicity that the size p of the input path is of the form 2^n. Level k contains $p/2^k$ clusters where each cluster index comprises 2^k set bits which define its $partial\_size$ (line 1). The algorithm iterates through the clusters of the level and joins every two consecutive pairs of clusters with indexes cl_s and cl_t. Each cluster index has the first set bit at a relative position to its horizontal position; the cluster at horizontal position i has the first bit set in position $i * partial\_size$ (lines 3, 4) and will comprise $partial\_size$ bits set (lines 5, 6). The join result of cl_s and cl_t is stored in the cluster with id $cl\_s \oplus cl\_t$ (line 7). Then, the algorithm iterates through all the sub-paths of the hash table of cl_s and probes the neighboring vertices for a match in the hash table of cl_t. Finally, it merges the selected sub-paths and stores them in the hash table of the resulting radix cluster (lines 11-13).

We scale-up the Join& Merge phase by assigning a worker thread to a subset of a cluster, or to a set of clusters of the radix cluster tree depending on the number of available threads. Given path size p, $p/2^k$ nodes at level k, and T threads, each thread will process $T \cdot 2^k/p$ nodes. Then, each thread is responsible for joining the subset of the clusters it is responsible for with their adjacent cluster.

The Join&Merge operation at each level follows the associative property because Join is an associate operator. Hence, the correctness of the parallel execution is equivalent to the correctness of the sequential execution. In addition, given that two or more threads might work on the same cluster, they will also end up updating the same hash table. However, there is no need for synchronization, as the two threads will work on non-overlapping positions of the hash table. Specifically, as the sub-paths are indexed based on the first vertex of a sub-path, in the case where two threads work on the same cluster, they will be responsible for paths with a different starting vertex. Thus, as the sub-paths expand by merging the end of the sub-path, the resulting sub-path will be indexed on the same hash table position as the left side of the join, *causing no collisions*.

The example shown in Figure 5.2, demonstrates the second phase of the path matching strategy. Join&Merge requires two iterations on the radix cluster tree of Figure 5.3 which follows the example of Figure 5.2. In the first iteration (first level of the tree), it joins the clusters with indexes 1000 and 0100 as well as 0010 and 0001. The merged result of the corresponding joins is stored into clusters 1100 and 0011 respectively. Similarly, the second iteration works on the topmost layer of the tree and joins clusters 1100 with 0011 which produce the final result as it has all bits set to 1.

Theorem 1. *The path matching algorithm that uses the radix cluster tree approach is sound and complete.*

Proof. To prove the soundness of the algorithm we must prove that any sub-path that the algorithm returns is correct. We will prove the soundness of Algorithms 4, 5, 6 by induction. Let us assume an input path of size $p \in \mathbb{N}$. The algorithm will produce $L = log_2 p + 1$ levels.

Base case: First level of the tree, $l = 1$. The first level discovers all single vertex paths that

qualify a given label. This process relies on the filtering selection of the subgraph isomorphism algorithm. Therefore we assume that the first phase which is responsible for the first level of the tree is sound based on the soundness of the subgraph isomorphism algorithm.

Inductive step: Show that for any level $l < L$, if the algorithm correctly identifies subpaths of size l, then it can correctly identify a subpath of size $l+1$. At level $l+1$ the Join&Merge algorithm will join every two adjacent clusters from level l. Thus, it will join only clusters with corresponding radix cluster indexes of the regular form $\{0^*1^+0^*\}^p$ such that the first subpath has the last 1 bit in position $pos < (p-1)$ of the path and the second subpath has the first 1 bit in position $pos+1$. This ensures that the second subpath is a candidate continuation of the first subpath because it contains a qualifying subpath starting from a consecutive position after the end of the first subpath. Apart from the fact that the subpaths are of complementary positions, we need to also prove that the edge connecting the partial paths qualifies the path. However, the *qualifying* method executed at the filtering step of the first phase checks the criteria that ensure that only qualifying neighbors connected with a qualifying edge are stored in the hash table. Therefore, based on the assumption that the feasibility criteria of the *qualifying* method of the subgraph isomorphism algorithm are correct, we assume that the resulting subpaths produced and stored at $l+1$ are also correct.

Conclusion: Since both the base case and the inductive step have been proved as true, by mathematical induction correctness holds for all subpaths/paths of every level $l \leq L$.

To prove the completeness of the algorithm, we must prove that the algorithm returns all qualifying paths, i.e., there is no qualifying path that the algorithm fails to return. Next, we provide the completeness proof for both phases of the algorithm. Similarly to the soundess proof, as the first phase relies on the completeness of the filtering selection of the subgraph isomorphism algorithm, then we assume that the first phase is complete. For the second phase, let us assume that there exists a valid path that the algorithm fails to include in the result set. Such a scenario would occur if the Join&Merge phase fails to join two subpaths belonging to radix clusters with indexes the form $\{0^*1^+0^*\}^p$ where the first subpath has the last 1 bit in position pos and the second has the first 1 bit in position $pos+1$. Such a case would happen if a cluster of the tree is missing which means that at least one of its parents are missing. Given that the first level of the radix tree contains all single-vertex paths due to the completeness of the first phase, such a case would happen if an intermediate level fails to join two complementary paths. However, based on the resulting binary radix tree that the algorithm incrementally builds, all complementary nodes are the adjacent nodes of each level. As the adjacent nodes are the ones that the algorithm chooses for a join, failing to join complementary nodes is impossible. Hence, as at each level all possible results get generated, the final set of paths will also be complete. □

Theorem 2. *The complexity of the search algorithm is $\mathcal{O}(|V|^{2\log p})$, where $|V|$ is the amount of vertices and p is the size of the query pattern.*

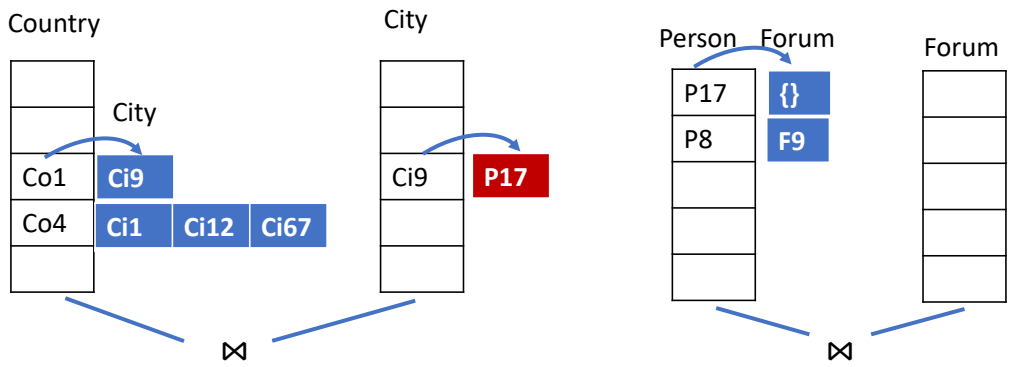


Figure 5.4 – Example of the progressive pruning of invalid partial states while joining states

Proof. Suppose that $p = 2^K$. In this case, the algorithm builds a binary tree where each node is identified using p bits which result by applying the bitwise XOR on the identifiers of its children. The leaves of the tree have only one bit set to 1, and this bit is different for every leaf. Accordingly, the tree will have p leaves and $K+1$ levels. Let the leaves be at level 0 and the root at level K. Every node of the tree indexes the set of candidate vertices and edges that match the query pattern at the positions where the bits of the node identifier are set. We refer to these sets as partial answer sets. It follows that the root of the tree indexes the full query answers.

In the worst case, all the partial answer sets will have the same size. Moreover, at every step $k \in [0, K-1]$, the algorithm merges all the pairs of partial answer sets and in the worst case (fully connected graph), the size of the result indexed by their parent will be equal to the product of the size of the merged sets. Therefore, each level has $\frac{p}{2^k}$ partial answer sets and each set has $\frac{|V|}{p}^{2k}$ candidate matches. By summing for all the levels:

$$\begin{aligned}
&\sum_{k=0}^{K-1} \frac{p}{2^k}\left(\frac{|V|}{p}\right)^{2k} = 2^K + \sum_{k=1}^{K-1} \frac{2^K}{2^k}\left(\frac{|V|}{2^K}\right)^{2k} = \\
&2^K + \sum_{k=1}^{K-1} 2^{-k(2K+1)+K}|V|^{2k} \leq 2^K + \sum_{k=1}^{K-1} |V|^{2k} = \\
&2^K + \frac{|V|^{2K} - |V|^2}{|V|^2 - 1} \leq 2^K + |V|^{2K} - |V|^2 \leq |V|^{2K}
\end{aligned} \tag{5.1}$$

As $K = log_2 p$, we conclude that $\mathcal{O}(|V|^{2\log p})$.

□

5.3.4 Progressive filtering optimization

The construction and merge of the clusters eliminates redundant checks by restricting each worker to a specific set of positions of the path. However, as each thread operates separately, it might happen that some threads produce large intermediate results which will be pruned afterwards. To avoid large intermediate results during the construction and merge of the clusters, we introduce a progressive filtering phase that allows transfering information from one worker to another in order to prune non qualifying intermediate results. More specifically, as the indexing of vertices that belong to different clusters takes place in parallel, if a partial state starting from a specific vertex fails to find a valid continuation, the filtering process marks it as invalid. Then, any thread that produces a state involving an invalid continuation marked by another thread, it will directly prune the state.

An example of the progressive filtering optimization is shown in Figure 5.4. While joining and merging Person and Forum vertices it appears that the person vertex with label *P17* has no valid match. Therefore, as the join of *P19* takes place before the join of country with label *Co1*, *Co1* will try to merge *Co1* with city *Ci9* but it will detext that *Ci9* has no valid continuation and it will prune the state.

The progressive filtering is sensitive to the order of traversing the radix clusters for join because vertices with high degree are the ones causing intermediate result explosion [46]. Therefore, defering their evaluation increases the knowledge obtained before producing large intermediate results. As the join takes place by joining vertices that appear in consecutive positions in the input path, we consider as high degree vertices the ones that have a high number of qualifying neighbors that appear in a consecutive position in the path; we will refer to the number of qualifying neighbors that constitute a possible continuation of a vertex with respect to the input path as the outer degree of the vertex. Apart from defering vertices with many possible continuations, we join early vertices that have multiple neighbors in the preceding position in the path. By this, we increase the impact of pruning an invalid vertex. By combining the above metrics, we order the vertices we define their rank based on the following formula:

$$Rank_{vi} = out\_degree(vi) - in\_degree_{(}vi)$$

As sorting the vertices based on the above ranking would introduce a significant overhead, we approximately order them by range partitioning the hash table based on the rankings that appear in the data. The boundaries of the ranges are derived from a random sample of the vertex ranks. Specifically, we extract a random sample of k vertices that have a candidate continuation and we sort them based on their $Rank_{vi}$. Then, by using the sorted set of ranks, we assign each vertex to the appropriate range based on its rank. By this we obtain a clustering of vertex ranks based on the number of outer and inner vertices they have.

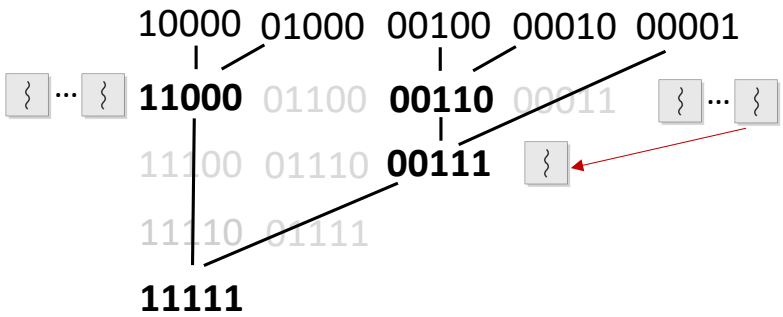


Figure 5.5 – Generalized example of radix tree traversal. When a thread terminates processing a node, it moves to the next level (red arrow) if both sides of the join have been built. Otherwise, it steals work from the threads working at the same level.

5.4 Radix tree generalization & Optimizations for load-imbalance

Algorithms 5, 6 are limited to the case where the size of the path is of the form 2^K. Moreover, parallelization takes place at each level of the tree, hence, the runtime is determined by the slowest cluster of the level. To address these issues, this section presents the generalized path matching algorithm, as well as a set of optimizations that address the limitations of staged execution and load-imbalance.

5.4.1 Generalized radix-tree traversal

In the following, we present the general algorithm for the path matching when the assumption that the path size is of the form 2^K does not hold for the leaf level. In that case it might happen that one node of a given level remains unmerged, hence when joining pairs of nodes of level l_i, we also need to check for every level $l_j < l_i$ whether there exists a node that qualifies for a join with any remaining node of level l_i. An example of a general radix tree for any path size is shown in Figure 5.5. In the first level of the tree there are five nodes, therefore two joins get executed. As we observe, the node 00001 is not selected for a join at the first level. Therefore, the second level will check whether any of the nodes can be joined with node 00001. As shown in the figure, node 00110 qualifies for joining with node 00001. When such a scenario occurs, a join might take place among nodes of different levels. For example, node 11000 will be joined with node 00111. To allow such cases, when a node n looks for a complementary node m to merge its partial state, out of the complementary candidate nodes, it will select the built node that belongs to the lowermost level of the tree. Therefore, the final radix cluster tree structure allows for a hybrid breadth-first-search and depth-fist-search traversal because it ends up merging partial paths by either expanding with the qualifying neighbors, or qualifying sub-paths respectively.

To optimize the radix tree construction through the join and merge traversal, we also avoid a strictly staged execution at each level, as this limits parallelization and thread utilization, because a thread has to wait until all threads have finished processing the partitions of the

level in order to be able to proceed to the next level. For example, in the case of Figure 5.5, if node 00110 has been computed before 11000, then a thread can already proceed and join it with node 00001 without having to wait for the thread responsible for node 11000 to finish. Therefore, we set every thread to iteratively check the tree level-by-level and join two nodes if and only if their parent nodes have been finalized. Even though this requires synchronization among the threads to ensure that no two threads select the same pair of nodes to join, the overhead is negligible.

5.4.2 Work-stealing for load-imbalance

Vertices in real-world graphs appear with different degrees, that is the number of neighbors of each vertex might vary significantly. What is more, when partitioning a graph in a workload-agnostic way there exist partitions with many qualifying vertices, whereas others will not have any. Clearly, this leads to load imbalance, which limits the effect of parallelization, as some threads finish when others still have work to do. Thus, even if some threads can continue the work by moving to the next level of the tree in the case where both parent hash tables have been built (Figure 5.5), this is not always possible and parallelization is limited at each level. The existence of load imbalance might affect both phases of the path matching evaluation.

Distributed algorithms typically approach the load imbalance problem by investing in a load-aware optimal partitioning approach. A load-aware partitioning approach either operates offline by optimizing partitioning algorithms towards equi-sized partitions [122, 84], or it operates at query time by continuously re-evaluating the load as it executes a query path [121]. An offline partitioner is, however, workload-agnostic and therefore fails to address the imbalance induced by the difference in the set of qualifying subgraphs in each partition. On the other hand, online partitioners add a significant overhead at each step of the path evaluation to balance the load. However, such online approaches are meaningful for distributed graphs, because remote partition accesses require communication over the network. The scale-up setting does not suffer from such overheads due to having a shared state, and therefore, there is need for a lightweight online balancing approach that determines the partition boundaries for each worker thread.

To mitigate the load imbalance problem while scaling up query processing, instead of having a static partitioning at each tree level, we dynamically assign work to threads through work-stealing. When threads finish their tasks they move to further partitions. Work-stealing applies to both phases of the processing by resizing the partition boundaries online. In the first phase work-stealing operates over the input graph, whereas in the second, it operates over the hash tables representing the nodes of the radix tree.

When a worker thread terminates, then it randomly picks another worker thread with work to do. In the case where it finds another worker with a higher load, then, depending on whether the execution is in the first phase, or in the Join&Merge phase, it computes the number of remaining vertices or sub-paths respectively that it has to process. The remaining vertices

are uniformly redistributed among the worker responsible for the partition and the new free worker. This process continues until all workers terminate with their partitions and the partitions they have stealed. By this, we ensure that no thread remains idle, and that we fully utilize the available threads for the path matching process.

We combine the level bypassing presented in Section 5.4.1 with work-stealing and we consider them as mechanisms to dynamically assign work to threads. Therefore each worker that looks for work will prioritize passing to the next level of the tree in case the hash tables of both clusters have been built. If it fails to find such pair of clusters, it will choose to steal work from the threads working at the current level. The prioritization takes place because the work-stealing requires synchronization and re-computation of partition boundaries among the involved threads, thereby incurring extra cost, whereas level bypassing is for free.

Summary: At each level of the radix tree there are two types of partitioning schemes that take place. The first scheme involves the initial partitioning presented in Section 5.3.3 which statically divides the clusters of one tree level across the workers; each worker might take as input a subset of a radix cluster, or even multiple radix clusters. The second scheme is the dynamic partitioning which employs work-stealing and level-bypassing in order to allow threads to look for work by either modifying on-the-fly cluster boundaries, or by dynamically discovering clusters to work on based on the load that each thread has.

5.5 Experimental Evaluation

In this section, we demonstrate the benefits of our approach in terms of efficiency and scalability, as well as how it compares with the state-of-the-art. In the following, we present experiments by (a) varying the size of the input paths, (b) the path complexity in terms of selectivity and degree of path vertices, and (c) the density of the dataset.

Datasets: We use two datasets to evaluate the efficiency of our system; one real-world dataset and one synthetic benchmark that has been built to evaluate graph data management applications. MusicBrainz [89] is a real-world music database that contains curated metadata of artists, their affiliations and their works. We use a subset of the dataset that includes information about music tracks, and we convert it into a graph. The resulting graph contains 8 distinct types of vertices and 7 distinct types of edges represented as labels, and in total contains 40 million vertices and 16 million edges. The LDBC Social Network Benchmark (SNB) [39, 3] is a synthetic dataset that contains a labeled graph that mimics the characteristics of a real-world graph. Specifically, the SNB dataset is a snapshot representing the activity of a social network. The graph includes vertex types such as Persons, Organisations, Posts, and Places. Edges between entities model interactions among Persons with other Persons, Places etc. Example relationships involve interests in posts, friendships, studies etc. We use two versions of the dataset, with scale factor 1 and 3 respectively. The resulting graphs contain 7 vertex types and 18 edges types which result in (a) 3 million vertices and 15 million edges for scale factor 1, and (b) 9 million vertices and 48 million edges. In both datasets we insert the

Dataset	#vertices	#edges	avg vertex degree	max degree
SNB	3M	15M	9	300K
SNB	9M	48M	9	300K
MusicBrainz	40M	17M	1	755

Table 5.1 – Characteristics of datasets

path	avg vertex degree	max degree
MusicBrainz 3	1	755
MusicBrainz 4	1	755
MusicBrainz 5	1	755
SNB 3	487	300K
SNB 4	107	300K
SNB 5	13	300K

Table 5.2 – Characteristics of paths

vertices by loading each type of vertex one after the other. Therefore when partitioning the graph, each partition will comprise vertices of one or more vertex types. The characteristics of the datasets are shown in Table 5.1.

Experimental setup: All experiments run on a node equipped with 2×Intel(R) Xeon(R) Gold 5118 CPU (12 cores per socket @ 2.30GHz), 64KB of L1 cache and 1024KB of L2 cache per core, 16MB of L3 cache shared, and 376GB of RAM. To exclude remote memory access overheads, we run all experiments in a single socket by using thread affinity.

We implement path matching in C++ and our implementation is based on the Boost Graph Library 1.65 for building and traversing the graph. The architecture employs an array that stores the vertices while the edge information is stored in the adjacency matrix.

We compare our approach with the parallel version of the baseline subgaph isomorphism algorithm VF2, our own optimized variation of VF2 that uses work stealing, and with Arabesque [121]. We also compare the second phase of our algorithm using MonetDB [2] and Empty-Headed [8] in order to evaluate the performance of the join and merge phases. In the following, we will be referring to our approach as *radix*, given that it is based on radix clustering to distribute the work. As a baseline approach we use the VF2 subgraph isomorphism algorithm which we scale up for a fair comparison. The VF2 algorithm [30] traverses the graph in a depth-first-search manner by maintaining and expanding a state that represents a qualifying sub-path. Specifically, starting from a qualifying vertex, the algorithm iteratively traverses the neighboring vertices and expands a state as long as it finds qualifying vertices. Then by using backtracking it identifies all possible qualifying substates. We modify the algorithm execution to map it to a scale-up setting by assigning each thread to a specific subset of the graph. Then, to avoid returning duplicate paths, we synchronize the work done by the threads by using two conflict resolution strategies: First, we allow a thread working on partition p_i to

continue expanding a partial state by accessing another partition, as long as it adds vertices that belong to any partition $p_j > p_i$ but not the other way around. Thus, we allow only the thread responsible for the smallest partition to continue the path computation. The second strategy avoids duplicate results within a partition. Specifically, within a partition when forming a qualifying path starting from a qualifying vertex, we only add vertices that have a higher vertex identifier (vertex id) than the starting vertex of the path. Therefore, in the case where a qualifying neighbor has lower vertex id, the candidate expansion is pruned and then the algorithm backtracks to the next candidate vertex. To evaluate the benefits of work stealing, we also implement a variation of the VF2 approach that uses work-stealing. The work-stealing idea is similar to our approach by allowing threads that have finished to steal a subset of the partition of another thread and compute all qualifying paths that involve the vertex of the partition.

We also compare our approach with Arabesque [121] which is a state-of-the-art graph mining system that was designed for a distributed setting. As the available implementation of Arabesque is built on top of Hadoop, to obtain a fair comparison, we port the implementation of Arabesque into our own implementation. Specifically, we implement the embedding exploration approach of Arabesque in a scale up setting: Initially, we execute the filtering phase where we discover qualifying vertices. Then, at the next iteration step, we pair the qualifying vertices with their qualifying neighbors. Similarly, at each iteration step, we expand the existing candidate embeddings with a neighboring qualifying vertex. We execute each embedding expansion by using multiple threads. To avoid duplicate paths, we apply the same coordination strategy with Arabesque where the lowest vertex id can expand an embedding. After each exploration step we execute the load-balancing that Arabesque employs in order to distribute the available embeddings across the workers. To balance the load, we consult the degree of the vertices that are to be expanded, and based on this information, we distribute the embeddings such as each partition obtains equal neighboring vertices to check.

5.5.1 Scalability with path size

In this experiment we compare the response time and the scalability of all approaches when varying the path size. We experiment with both the SNB and the MusicBrainz dataset in order to evaluate the efficiency in the case of both dense and sparse datasets; SNB is a dense dataset while MusicBrainz is sparse. We run the experiments by using 4 to 24 threads in order to observe the scalability of each approach. We limit the number of threads to 24 as it constitutes the maximum number of threads that can be placed in one socket. By this, we avoid any possible NUMA effects caused by remote accesses.

SNB: We evaluate the response time when executing paths comprising 3 to 5 edges. Specifically, we use the following paths: $Person \rightarrow_{studiesIn} Uni \rightarrow_{locatedIn} City \rightarrow_{isPartOf} Country$, $Forum \rightarrow_{hasModerator} Person \rightarrow_{studiesIn} Uni \rightarrow_{locatedIn} City$ $\rightarrow_{isPartOf} CountryPost \rightarrow_{partOf} Forum \rightarrow_{Moderator} Person$

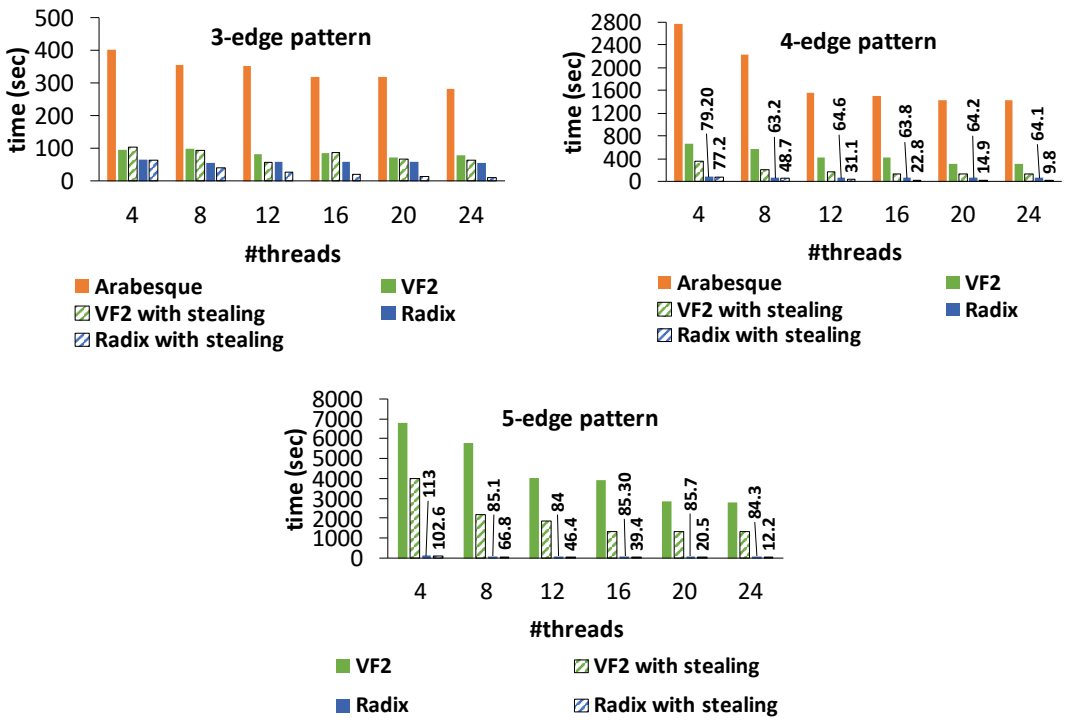


Figure 5.6 – Response time for paths comprising 3, 4, and 5 edges respectively for SNB.

$\rightarrow_{studiesIn}$ Uni $\rightarrow_{locatedIn}$ $City$ $\rightarrow_{isPartOf}$ $Country$. The characteristics of the paths in terms of vertex degree are shown in Table 5.2.

Figure 5.6 shows the path matching response time for the SNB dataset in the case of the aforementioned paths with 3, 4, and 5 edges respectively. We observe that in all cases, the radix variations are faster than both the VF2 variations as well as Arabesque. The main difference of radix with VF2 stems from the fact that to coordinate the work among the threads, VF2 drops a partial path when a vertex with vertex id lower than the first qualifying vertex of the path is added. On the other hand, in the approach that uses the radix clustering, each worker is responsible for matching specific positions in the path thereby avoiding redundant work. In the case where the path is smaller, i.e., 3-edges, then the difference between the VF2 approach and radix clustering is smaller (~1.5×) because there is a low probability of having two threads computing the same path, therefore the duplicate work that VF2 performs is negligible. The main reason of having a speedup over VF2 is due to the filtering pass that prunes the non-qualifying neighbors of the qualifying vertices. By filtering out non-viable paths, our approach avoids to re-iterate through the neighbors of a qualifying vertex in the case where the vertex participates in multiple paths. On the other hand, in the VF2 approach, as it proceeds by using depth-first-search when a vertex v participates in two different paths $path_1$ and $path_2$, to continue the path after v the algorithm needs to check the neighbors of v twice, once for each path.

In the case of paths comprising 4 and 5 edges we observe that the difference between VF2 and clustering is more significant: ~6× for the 4-edge path and ~20× for the 5-edge path. In this case the benefit stems from two reasons: First, VF2 executes more redundant work when the size of the path increases; it might prune sub-paths late in their evaluation in the case where the next vertex that is added is smaller than the starting vertex of the sub-path. Second, as with the 3-edge paths, the filtering phase of the radix clustering approach filters out non-qualifying neighbors and uses the hash tables to index only the qualifying continuations of each partial sub-path.

We also observe in Figure 5.6 that Arabesque performs poorly in all cases because it traverses the graph at each level of expansion in order to distribute again the embeddings across the workers. To do this, it checks the degrees of the vertices to be expanded in order to estimate the amount of work that each worker thread will have. Having to iterate and repartition the embeddings at each step is expensive and slows down the overall execution. For the path comprising 5 edges arabesque was unable to respond after a timeout of 3 hours, therefore we omit it from the graph.

Work stealing benefit. Between the vanilla VF2 and the work stealing variation of VF2, the work-stealing starts showing a benefit at the 4-edge paths where the distribution of the vertices of the path is more skewed; the forum vertex appears in the 4-edge path, and as it has higher selectivity than the other vertices (10× higher), it slows down the execution of the threads responsible for the forum vertices compared to other threads responsible for partitions with fewer qualifying vertices. The case is similar for the 5-edge path which also contains the forum vertex. In the case of the radix clustering, the benefit of the work stealing is evident regardless of the size of the path; the work-stealing variation is on average 3× faster than the pure radix clustering approach. The reason for this is that in the radix case, load imbalance dominates the overall execution as the other bottlenecks have been eliminated. However, in the VF2 case, iteratively checking the validity of the neighbors, as well as executing redundant work is the main source of showing poor performance. We also observe that in all cases, the difference between the simple radix and the radix variation with work-stealing increases as we increase the number of threads. The reason is that when we increase the parallelization, there are more workers that have lower load and therefore they can steal work from other threads. Whereas when there are few threads there is a higher probability of having similar load and therefore show fewer opportunities for stealing.

Summary. The radix approach scales better than state-of-the-art approaches because it distributes the work evenly across the worker threads; it distributes both the work required for evaluating a single path by eliminating any redundant work, as well as the work across paths by using work stealing.

MusicBrainz: In the case of the MusicBrainz dataset, we use three different query types which we derive by executing prefix paths of the following path query: $Series \rightarrow_{hasRecording} Recording \rightarrow_{hasTrack} Track \rightarrow_{hasMedium} Medium \rightarrow_{hasRelease} Release \rightarrow_{hasReleaseGroup}$.

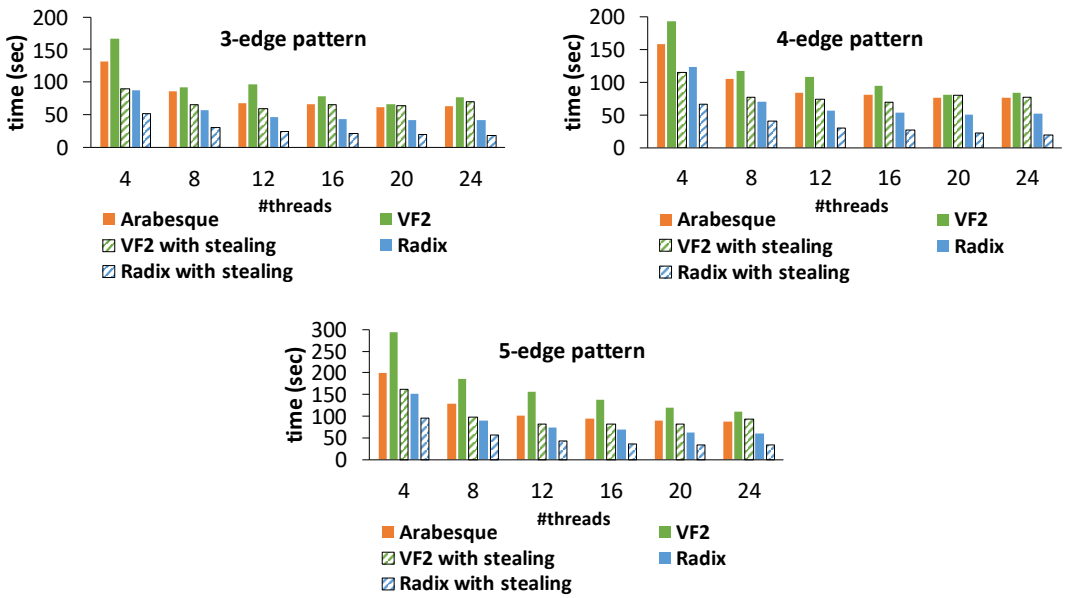


Figure 5.7 – Response time for paths comprising 3, 4, and 5 edges respectively for the MusicBrainz benchmark.

Specifically, we execute path queries comprising 3, 4, and 5 edges respectively, in order to obtain information about a series of recordings. The characteristics of the paths are shown in Table 5.2. MusicBrainz is a sparser dataset than SNB, but it is also skewed in terms of vertex degree; even though on average each vertex has one neighbor, the $Track$ vertex which is the most popular, has up to 700 neighbors.

Figure 5.7 shows the resulting execution times for all approaches in the case of the aforementioned path queries over MusicBrainz. We measure the response time by varying the number of threads from 4 to 24. We observe that the difference between the radix variations and the VF2 ones is smaller than in the case of SNB. The reason is that MusicBrainz is a significantly sparser dataset than SNB because the vertices have a lower degree and therefore when VF2 computes the qualifying paths it has less work when checking the neighbors of each qualifying vertex. However, still VF2 performs duplicate work and therefore the radix outperform VF2 (~1.8× on average). We also observe that in all paths, the work-stealing benefit in the case of VF2 decreases as we increase the number of threads. The reason is that as we increase the number of threads the duplicate work becomes the bottleneck and therefore the difference due to the skew becomes less important. However, when considering the radix variations, work-stealing is always beneficial (~ 1.7× on average) due to the indexing that takes place at each level which causes overloaded partitions. Specifically, the clusters involving the $Track$ vertex cause load imbalance and the threads responsible for the other vertices steal work to balance the work.

Due to the presence of several vertices with low degree in the case of MusicBrainz, Arabesque

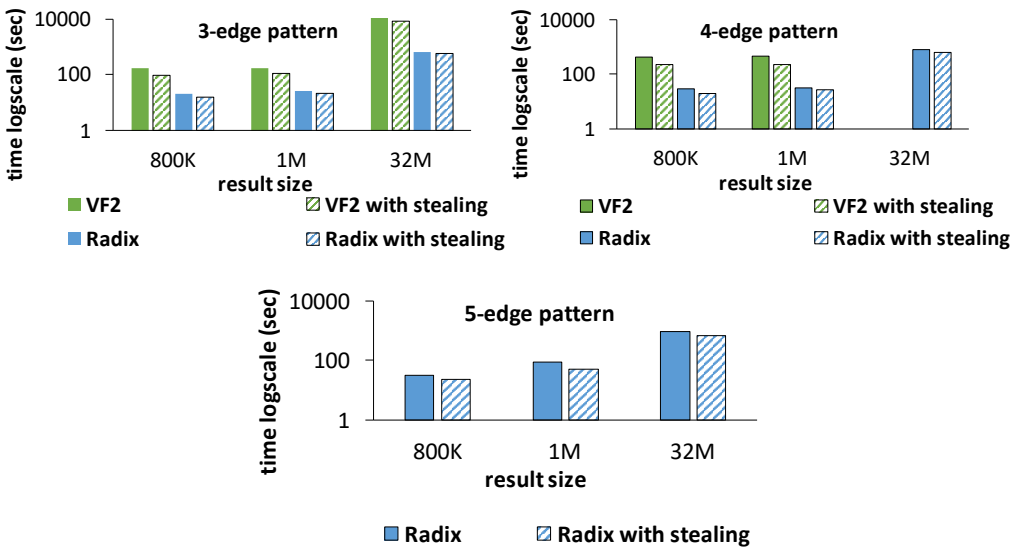


Figure 5.8 – Response time for varying path with 3, 4, and 5 edges respectively.

also performs better, compared to the SNB case. In addition, Arabesque performs better than the VF2 approach in all cases which justifies its design for balancing the load. Arabesque also follows the same trend with VF2 work stealing by having a less significant impact when the number of threads increases because it also performs extra work by pruning candidate duplicate paths. However, Arabesque is slower than VF2 work-stealing because it requires multiple passes over the data to redistribute the load. Arabesque is also slower than the radix variations due to both the duplicate work it requires, as well as due to that it iteratively traversing the neighbors of the qualifying vertices.

5.5.2 Varying path complexity

In this section we evaluate the response time of all systems when varying the complexity of the path. We follow a similar setup where we execute the path matching process using variations of paths with 3, 4, and 5 edges respectively. The variations involve both different path selectivities as well as paths with different vertex degrees. We execute the following three paths and their subsets to form paths of size 3, 4, and 5 edges: (a) $Post \rightarrow_{containedIn} Forum \rightarrow_{hasMember} Person \rightarrow_{studyAt} Uni \rightarrow_{isLocatedIn} City \rightarrow_{isPartOf} Country$, (b) $Comment \rightarrow_{replyOf} Post \rightarrow_{containedIn} Forum \rightarrow_{hasModerator} Person \rightarrow_{isLocatedIn} City \rightarrow_{isPartOf} Country$, (c) $Comment \rightarrow_{replyOf} Post \rightarrow_{containedIn} Forum \rightarrow_{hasModerator} Person \rightarrow_{studyAt} Uni \rightarrow_{isLocatedIn} City$. The path queries and the subsets of them return 800K, 1M, and 32M qualifying sub-paths respectively. The degree of the paths ranges from 7 to 12 neighbors on average when going from 800K to the 32M case. We execute the path matching process by using 24 threads in all cases.

Figure 5.8 shows the response time by considering different combinations of the above paths.

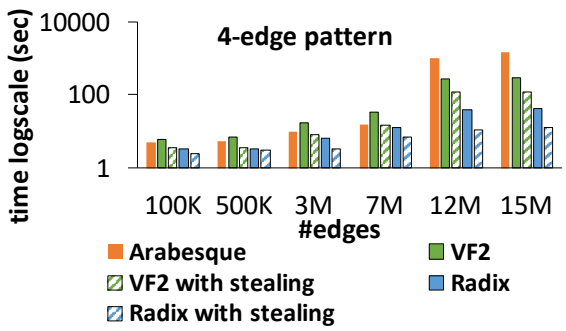


Figure 5.9 – Response time when increasing the vertex degree.

In the case of the 5-edge paths, VF2 was non-responsive, therefore we only report results for the radix-based approach. There are several observations to be made. First, in the 32M case, which has the highest selectivity, the response time is significantly higher than the 1M and 32M case regardless of the approach or the size of the path. Similarly, the 1M case is slightly slower than the 800K case as it involves the *Comment* vertex which appears more frequently in the graph. Therefore, the selectivity of the path is crucial for the overall execution time even if we consider paths of similar vertex degrees. Further, the radix variations are much less severely affected by the path selectivity compared to the VF2 case. The reason of having better scalability in the radix approach is that, by indexing the partial states, we avoid any redundant work which dominates in the case of vertex types with high selectivity and/or high degree.

5.5.3 Varying vertex degree

In this experiment we measure the response time for all approaches when increasing the degree of the vertices. To observe the effect of the vertex degree on performance, we keep the number of vertices and the result size fixed, and we increase the number of edges that do not affect the result, i.e., we modify the number of edges that do not participate in the path. Then, we execute the path matching process using 24 threads.

Figure 5.9 shows the response time as we increase the total number of edges of the dataset from 100K to 15M for the path that comprises 4 edges. We observe that for all systems, the response time increases as we increase the degree. In the case of 12M edges there is a significant increase in response time because the added edges increase the degree of the *Place* vertex which participates in the path. Therefore, it increases the difficulty of the path evaluation in all cases. In the case of the radix variations, the first level that iterates and prunes the neighbors is affected because afterwards radix clustering operates over the qualifying part of the graph which remains fixed. Similarly, VF2 and Arabesque also have to iterate multiple times through the neighbors therefore, the increase in response time is significantly higher than in the case of radix.

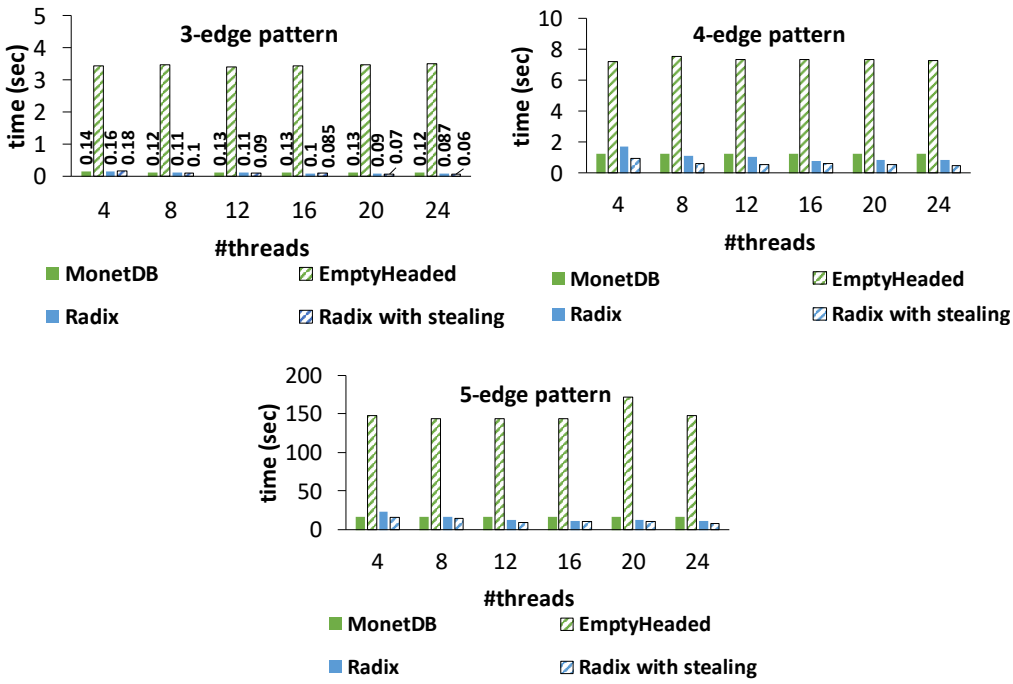


Figure 5.10 – Response time for varying path with 3, 4, and 5 edges respectively.

5.5.4 Join&Merge phase evaluation

In this experiment we measure the response time and scalability of the Join&Merge phase compared to MonetDB and EmptyHeaded. We use in all cases SNB with scale factor 3 this time. The result of the experiment is shown in Figure 5.10. We observe that radix with stealing scales better than both MonetDB, as well as EmptyHeaded. The reason is that it exploits both the work stealing to address skew and the lightweight pruning of intermediate results. MonetDB has stable response time as we increase the number of threads because the threads responsible for the high degree vertices slow down the overall join execution; the high degree vertices produce a larger number of results. EmptyHeaded has a high pre-processing cost due to the high response time of optimization and code generation. As a result, even though EmptyHeaded exploits SIMD at the generated code the main bottleneck is the optimization and code generation phases, therefore the overall response time remains the same when increasing the number of threads.

5.5.5 Optimization for large intermediate results

In this experiment we measure the benefit of the progressive filtering optimization to prune large intermediate results. We execute the path matching over the SNB dataset with scale factor 3. We use a path consisting of 5 edges that has the following form: $Post \rightarrow_{containedIn} Forum \rightarrow_{hasMember} Person \rightarrow_{studyAt} Uni \rightarrow_{isLocatedIn} City \rightarrow_{isPartOf} Country$. As a Fo-

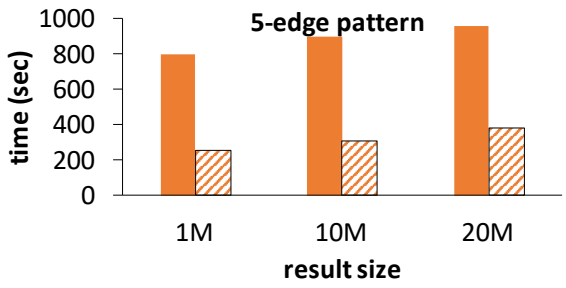


Figure 5.11 – Response time with and without using the progressive pruning optimization to prune online invalid partial states.

rum has multiple members, this relationship introduces has high selectivity due to the high degree of the Forum vertex. To measure the benefit of pruning invalid persons that mistakenly appear in the result of the $Forum \rightarrow_{hasMember} Person$ relation we tune the $studiesAt$ relationship in order to vary the selectivity of the path matching. Specifically the number of qualifying subgraphs ranges from 1M to 20M.

Figure 5.11 shows the response time as we increase the number of qualifying subgraphs. We measure the response time when we activate the progressive pruning strategy by approximately sorting based on the vertex degree compared to executing the traditional join and merge without the on-the-fly filtering. We observe that in the case of lower selectivity, the difference in response time when we activate the progressive pruning optimization is higher. The reason is that there are more opportunities for pruning; invalid pairs get pruned earlier and therefore the intermediate join result is significantly smaller.

5.6 Summary

Graph path matching is a task that has become widely used in graph analysis as it allows extracting useful information in a broad range of areas starting from social networks to scientific data. Due to the high complexity of the path matching problem systems employ parallelization to reduce the cost. However, applying preprocessing to optimally partition the work across different workers induces a significant overhead.

Our work introduces a system that allows parallel path matching over labelled graphs without the need for any preprocessing. We optimize path query evaluation in order to eliminate redundant work using radix trees. We also balance the work among the available workers by applying work-stealing. Our approach scales better than existing path matching approaches and can handle paths over dense graphs that other approaches are unable to support.

6 The Big Picture

Data cleaning is a challenging, yet indispensable pre-processing part of data analysis. Detecting and repairing errors over unknown data requires many iterations of exploring, understanding, and analyzing data, which further impedes insight extraction. Existing approaches that attempt to automate the data cleaning procedure have three main shortcomings: (a) They focus on a specific use-case and operation, (b) they either are offline by operating before analysis begins, or they require expensive pre-processing to cover complex rules, and (c) they cover only tabular data. First, by having systems that focus on a specific operation, from a user's perspective, one is forced to use a different, potentially inefficient tool for each category of errors. Furthermore, having to apply -each time- each cleaning task over the whole dataset is tedious and time-consuming. Still, even such specialized, offline tools exhibit long running times or fail to process large complex datasets.

In this thesis, we have set up the stage for efficient and real-time data cleaning that takes into consideration the type of operation, the analysis, and the data format. To achieve this goal, we designed the data cleaning stack by providing primitives that enable for a unified representation and optimization of the building blocks of different cleaning operations, while integrating them with exploratory analysis.

In this chapter, we summarize the contributions of this thesis and we discuss a number of interesting research topics related.

6.1 What We Did: Data and Workload-Aware Cleaning

We have identified two dimensions that must guide the design of data-cleaning systems: (a) coverage of the data cleaning operations and the underlying data formats and (b) awareness of data analysis tasks. In each part of this thesis, we redesigned and broadened data-cleaning systems by providing the appropriate abstractions that address the challenges.

To address the coverage issue of data cleaning, we have introduced CleanM, a declarative query language that enables users to express their specific cleaning scripts. CleanM exposes a wide variety of parameterizable data cleaning primitives that users can apply over their 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 implemented 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 or are unable to serve due to performance issues.

To enable online data cleaning, we have introduced Daisy: a system that cleans only the part of the dataset required by the exploratory queries that users perform. Daisy optimally integrates cleaning operators inside the query plan and efficiently executes them over dirty data by providing probabilistic answers for the erroneous entities. Our evaluation has shown that CleanDB adapts to the workload and outperforms traditional offline cleaning on both synthetic and real-world workloads.

To support and optimize cleaning building blocks for complex data types, we have introduced a system that enables parallel pattern-matching over labeled graphs, without the need for pre-processing. We optimized pattern query evaluation over graphs (a) by eliminating redundant work by using radix trees and (b) by progressively pruning intermediate results at pattern evaluation time. Our approach also addresses the load-balancing problem that constitutes a major issue in graph data due to the variability of both vertex degrees, as well as qualifying areas. To balance the work among the available workers we apply work-stealing at execution-time. Our approach scales better than existing pattern matching approaches and can handle paths over dense graphs that other approaches are unable to support.

6.2 Looking Ahead: Data and Workload-Aware Cleaning

The design of the data cleaning architecture provides the opportunity to support data cleaning tasks with different representation and optimization requirements. In the following section, we present possible extensions that are based on our design.

6.2.1 Cleaning Errors in Data Streams

The process of detecting inconsistencies requires different low-level optimizations, depending on the cleaning operation. For example, duplicate detection which involves similarity checks could be optimized by pruning comparisons through indexing, blocking, or clustering. in order to prune the comparisons. The filtering problem for fuzzy joins has been extensively discussed in both a single-node, as well as a distributed setting [61, 111, 119, 28, 26]. However, existing solutions are insufficient for a streaming setting. In a data streaming scenario, the resulting indexes or clusters need to be stored and continuously updated in order to enable the elimination of duplicates in future events. By continuously collecting and storing more data in the clusters, the stored information might exceed the memory capacity. Therefore, having to store, access, or even to process clusters or large indexes of historical data violates the

low-latency requirements of streaming applications. Existing solutions for the load-imbalance problem in distributed data streams are inappropriate as they assume apriori knowledge of the load. Moreover, having to pay the rebalancing cost each time is inefficient.

To address the problem of maintaining and accessing large states of clusters or indexes of data, there is a need for both compression techniques, as well as sophisticated data placement in storage devices in order to efficiently access hot data. Therefore, in order to enable efficient access and processing, there is a need for predicting which blocks of data will be used more often or that will be used subsequently.

To solve the load-balancing problem, one possible direction is to divide the set of similar blocks in such a way that both the workers have an equal number of tuples, while at the same time there is minimum work required to merge the work done by each subset of the block. Moreover, the repartitioning must induce a negligible overhead, as it will take place online. Therefore, proposing a solution for the load balancing problem involves introducing the appropriate heuristics, as it is an NP-hard multi-objective optimization problem.

6.2.2 Complex Cleaning Operations over Graphs

Optimizing cleaning operations over graphs is also challenging both in a scale-up as well as in a scale-out setting. Detecting patterns with similar labels also involves the expensive pattern matching task but also has the overhead of similarity comparisons. Therefore, combining the two operations by identifying candidate similar patterns while discovering them is one direction to interleave the two operations. Moreover, in a distributed setting, the similarity check process might end up shuffling patterns among the nodes. To address this issue, the partitioning needs to assign candidate similar entities in the same partition. Consequently, enabling and optimizing the similarity-aware partitioning is challenging, because it involves the extra overhead of estimating candidate similar patterns.

6.2.3 Optimizing Data Cleaning for Machine-Learning Workloads

Several machine-learning models, such as neural networks, require a vast amount of training data, some of which can be unrelated to the domain of the analysis. For example, several tasks propose pre-training models by using general-purpose data, and then the models can be fine-tuned using specialized data [35]. Even though the model requires accurate domain-specific data, it tolerates errors in the general-purpose data. By exploiting this property, we can introduce analysis-aware cleaning techniques and integrate error-tolerance inside an incremental cleaning approach for machine learning. Such a direction will be based on detecting semantic correlations among the detected dirty entities and on deciding which subset is crucial for the analysis. Thus, introducing noise-aware models that discover candidate dirty correlations concurrently to training can reduce the data cleaning cost, as well as the number of costly iterations for discovering inconsistencies.

Bibliography

[1] Historical air quality. https://www.kaggle.com/epa/epa-historical-air-quality.

[2] MonetDB. https://www.monetdb.org.

[3] The ldbc social network benchmark. https://github.com/ldbc/ldbc_snb_datagen.

[4] SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/, .

[5] SPARQL Property Paths. https://www.w3.org/TR/sparql11-property-paths/, .

[6] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 411–422. VLDB Endowment, 2007. ISBN 9781595936493.

[7] Ziawasch Abedjan et al. DataXFormer: A robust transformation discovery system. In *ICDE*, 2016.

[8] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4), October 2017. ISSN 0362-5915. doi: 10.1145/3129246. URL https://doi.org/10.1145/3129246.

[9] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia 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. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213864. URL http://doi.acm.org/10.1145/2213836.2213864.

[10] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB*, 2015. URL http://www.vldb.org/pvldb/vol9/p120-altwaijry.pdf.

[11] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, February 2018. ISSN 2150-8097. doi: 10.14778/3184470.3184473. URL https://doi.org/10.14778/3184470.3184473.

[12] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1421–1432, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3190654. URL https://doi.org/10.1145/3183713.3190654.

[13] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, New York, NY, USA, 1999. ACM. ISBN 1-58113-062-7. doi: 10.1145/303976.303983. URL http://doi.acm.org/10.1145/303976.303983.

[14] Michael Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.

[15] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. Messing up with bart: Error generation for evaluating data-cleaning algorithms. *Proc. VLDB Endow.*, 9(2):36–47, October 2015. ISSN 2150-8097. doi: 10.14778/2850578.2850579. URL http://dx.doi.org/10.14778/2850578.2850579.

[16] L. Berti-Équille, T. Dasu, and D. Srivastava. Discovery of complex glitch patterns: A novel approach to quantitative data cleaning. In *ICDE*, 2011. doi: 10.1109/ICDE.2011.5767864.

[17] Laure Berti-Équille, Ji Meng Loh, and Tamraparni Dasu. A masking index for quantifying hidden glitches. *Knowledge and Information Systems*, 44(2):253–277, August 2015. ISSN 0219-1377. doi: 10.1007/s10115-014-0760-0. URL http://dx.doi.org/10.1007/s10115-014-0760-0.

[18] Bibek Bhattarai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3300086. URL https://doi.org/10.1145/3299869.3300086.

[19] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915236. URL https://doi.org/10.1145/2882903.2915236.

[20] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009. ISBN 1586039296.

[21] Jens Bleiholder and Felix Naumann. Declarative data fusion – syntax, semantics, and implementation. In *ADBIS*, 2005.

[22] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999. URL http://www.vldb.org/conf/1999/P5.pdf.

[23] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *PVLDB*, 7(13), 2014. ISSN 2150-8097.

[24] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. Query optimization for dynamic imputation. *Proc. VLDB Endow.*, 2017. ISSN 2150-8097. doi: 10.14778/3137628.3137641. URL https://doi.org/10.14778/3137628.3137641.

[25] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1335–1349, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389742. URL https://doi.org/10.1145/3318464.3389742.

[26] Zhimin Chen, Yue Wang, Vivek Narasayya, and Surajit Chaudhuri. Customizable and scalable fuzzy join for big data. *Proc. VLDB Endow.*, 12(12):2106–2117, August 2019. ISSN 2150-8097. doi: 10.14778/3352063.3352128. URL https://doi.org/10.14778/3352063.3352128.

[27] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, April 2013. doi: 10.1109/ICDE.2013.6544847.

[28] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proc. VLDB Endow.*, 9(11):864–875, July 2016. ISSN 2150-8097. doi: 10.14778/2983200.2983203. URL https://doi.org/10.14778/2983200.2983203.

[29] Jeffrey Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2): 1481–1492, 2009.

[30] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct 2004. ISSN 1939-3539. doi: 10.1109/TPAMI.2004.75.

[31] Cypher. Cypher query language. https://neo4j.com/developer/cypher/.

[32] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, 2013.

[33] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003. ISBN 0471268518.

[34] Tamraparni Dasu and Ji Meng Loh. Statistical distortion: Consequences of data cleaning. *CoRR*, abs/1208.1932, 2012. URL http://arxiv.org/abs/1208.1932.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[36] Akhil A. Dixit. CAvSAT: A system for query answering over inconsistent databases. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, SIGMOD '19, pages 1823–1825, New York, NY, USA, 2019. ACM.

[37] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014. ISSN 2150-8097. doi: 10.14778/2732286.2732289. URL https://doi.org/10.14778/2732286.2732289.

[38] Orri Erling and Ivan Mikhailov. *RDF Support in the Virtuoso DBMS*, pages 7–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02184-8. doi: 10.1007/978-3-642-02184-8_2. URL https://doi.org/10.1007/978-3-642-02184-8_2.

[39] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742786. URL https://doi.org/10.1145/2723372.2742786.

[40] Wenfei Fan. Data quality: From theory to practice. *SIGMOD Record*, 44(3):7–18, December 2015. ISSN 0163-5808. doi: 10.1145/2854006.2854008. URL http://doi.acm.org/10.1145/2854006.2854008.

[41] Wenfei Fan and Ping Lu. Dependencies for graphs. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 403–416, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341981. doi: 10.1145/3034786.3056114. URL https://doi.org/10.1145/3034786.3056114.

[42] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16,

page 1843–1857, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915232. URL https://doi.org/10.1145/2882903.2915232.

[43] Leonidas Fegaras. Incremental query processing on big data streams. *TKDE*, 28(11): 2998–3012, 2016.

[44] Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. *TODS*, 25(4):457–516, December 2000. ISSN 0362-5915.

[45] Hugo Firth and Paolo Missier. Taper: Query-aware, partition-enhancement for large, heterogenous graphs. *Distrib. Parallel Databases*, 35(2):85–115, June 2017. ISSN 0926-8782. doi: 10.1007/s10619-017-7196-y. URL https://doi.org/10.1007/s10619-017-7196-y.

[46] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(12):1891–1904, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407797. URL https://doi.org/10.14778/3407790.3407797.

[47] Ariel D. Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005*, pages 337–351, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30570-5.

[48] Helena Galhardas. Data Cleaning and Transformation Using the AJAX Framework. In *GTTSE*, 2005.

[49] Helena Galhardas et al. Improving data cleaning quality using a data lineage facility. In *DMDW*, 2001.

[50] Stella Giannakopoulou, Manos Karpathiotakis, Benjamin Gaidioz, and Anastasia Ailamaki. Cleanm: An optimizable query language for unified scale-out data cleaning. *Proc. VLDB Endow.*, 10(11):1466–1477, August 2017. ISSN 2150-8097. doi: 10.14778/3137628.3137654. URL https://doi.org/10.14778/3137628.3137654.

[51] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606157.

[52] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. Cliquesquare: Flat plans for massively parallel rdf queries. In *2015 IEEE 31st International Conference on Data Engineering*, pages 771–782, 2015. doi: 10.1109/ICDE.2015.7113332.

[53] GraphX. Apache Spark GraphX. https://spark.apache.org/graphx/.

[54] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 491–500, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608044.

[55] Paolo Guagliardo and Leonid Libkin. Making sql queries correct on incomplete databases: A feasibility study. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4191-2. doi: 10.1145/2902251.2902297. URL http://doi.acm.org/10.1145/2902251.2902297.

[56] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465300. URL https://doi.org/10.1145/2463676.2465300.

[57] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 405–418, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376660. URL https://doi.org/10.1145/1376616.1376660.

[58] Yeye He, Kris Ganjam, Kukjin Lee, Yue Wang, Vivek Narasayya, Surajit Chaudhuri, Xu Chu, and Yudian Zheng. Transform-data-by-example (tde): Extensible data transformation in excel. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3193539. URL http://doi.acm.org/10.1145/3183713.3193539.

[59] Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends® in Databases*, 5(4):281–393, 2015. ISSN 1931-7883. doi: 10.1561/1900000045. URL http://dx.doi.org/10.1561/1900000045.

[60] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. Asap: Fast, approximate graph pattern mining at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 745–761, USA, 2018. USENIX Association. ISBN 9781931971478.

[61] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String Similarity Joins: An Experimental Evaluation. *PVLDB*, 7(8):625–636, 2014. ISSN 2150-8097. doi: 10.14778/2732296.2732299. URL http://dx.doi.org/10.14778/2732296.2732299.

[62] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *SIGCHI*, 2011.

[63] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014. URL http://www.vldb.org/pvldb/vol7/p1119-karpathiotakis.pdf.

[64] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper8.pdf.

[65] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12), 2016.

[66] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. BigDansing: A System for Big Data Cleansing. In *SIGMOD*, 2015.

[67] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, 26(1):125–150, Feb 2017. ISSN 0949-877X. doi: 10.1007/s00778-016-0441-6. URL https://doi.org/10.1007/s00778-016-0441-6.

[68] Won Kim. On Optimizing an SQL-like Nested Query. *TODS*, 7(3):443–469, 1982.

[69] Knime.

[70] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *Proc. VLDB Endow.*, 6(6): 397–408, April 2013. ISSN 2150-8097. doi: 10.14778/2536336.2536341. URL https://doi.org/10.14778/2536336.2536341.

[71] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[72] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[73] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[74] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9 (12):948–959, August 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994514. URL http://dx.doi.org/10.14778/2994509.2994514.

[75] D. Lasalle and G. Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236, 2013.

[76] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, December 2012. ISSN 2150-8097. doi: 10.14778/2535568.2448946. URL https://doi.org/10.14778/2535568.2448946.

[77] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, December 2012. ISSN 2150-8097. doi: 10.14778/2535568.2448946. URL https://doi.org/10.14778/2535568.2448946.

[78] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, November 2011. ISSN 2150-8097. doi: 10.14778/2078331.2078340. URL https://doi.org/10.14778/2078331.2078340.

[79] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, page 74–85, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307918. doi: 10.1145/2274576.2274585. URL https://doi.org/10.1145/2274576.2274585.

[80] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 225–237. ACM, 2018. ISBN 978-1-4503-4706-8. doi: 10.1145/3196959.3196980. URL http://doi.acm.org/10.1145/3196959.3196980.

[81] Steve Lohr. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights, The New York Times, 2014.

[82] Antonio Loureiro, Luis Torgo, and Carlos Soares. Outlier detection using clustering methods: a data cleaning application. In *IN PROCEEDINGS OF THE DATA MINING FOR BUSINESS WORKSHOP*, 2004.

[83] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI' 10, page 340–349, Arlington, Virginia, USA, 2010. AUAI Press. ISBN 9780974903965.

[84] L. Lu and B. Hua. Query-sensitive graph partitioner for pattern matching applications. *IEEE Access*, 7:184668–184675, 2019.

[85] Mohammad Mahdavi and Ziawasch Abedjan. Baran: Effective error correction via a unified context representation and transfer learning. *Proc. VLDB Endow.*, 13(12): 1948–1961, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407801. URL https://doi.org/10.14778/3407790.3407801.

[86] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807184. URL https://doi.org/10.1145/1807167.1807184.

[87] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, page 169–178, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132336. doi: 10.1145/347090.347123. URL https://doi.org/10.1145/347090.347123.

[88] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342643. URL https://doi.org/10.14778/3342263.3342643.

[89] MusicBrainz. Music brainz dataset. https://musicbrainz.org.

[90] neo4j. Neo4j graph database. https://neo4j.com/product/neo4j-graph-database/.

[91] Thomas Neumann and Gerhard Weikum. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453927. URL https://doi.org/10.14778/1453856.1453927.

[92] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989423. URL http://doi.acm.org/10.1145/1989323.1989423.

[93] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, pages 237–252, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10424-4.

[94] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *VLDB*, 9(9):684–695, May 2016. ISSN 2150-8097. doi: 10.14778/2947618.2947624. URL http://dx.doi.org/10.14778/2947618.2947624.

[95] Linnea Passing et al. Sql-and operator-centric data analytics in relational main-memory databases. *EDBT*, 2017.

[96] Paxata.

[97] Pentaho. Pentaho. http://www.pentaho.com/.

[98] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. Learning over dirty data without cleaning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1301–1316, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389708. URL https://doi.org/10.1145/3318464.3389708.

[99] Hasso Plattner and Alexander Zeier. *In-Memory Data Management: Technology and Applications*. Springer Publishing Company, Incorporated, 2012. ISBN 3642295746, 9783642295744.

[100] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx.iso: Parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329828. doi: 10.1145/2621934.2621939. URL https://doi.org/10.1145/2621934.2621939.

[101] Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *PVLDB*, pages 381–390, 2001.

[102] RAWLabs. RAW Labs. http://www.raw-labs.com/.

[103] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 2017. ISSN 2150-8097. doi: 10.14778/3137628.3137631. URL https://doi.org/10.14778/3137628.3137631.

[104] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, January 2015. ISSN 2150-8097. doi: 10.14778/2735479.2735493. URL https://doi.org/10.14778/2735479.2735493.

[105] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522740. URL https://doi.org/10.1145/2517349.2522740.

[106] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 410–424, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815408. URL https://doi.org/10.1145/2815400.2815408.

[107] Sudeepa Roy. *Uncertain Data Lineage*. Springer New York, 2018. ISBN 978-1-4614-8265-9. doi: 10.1007/978-1-4614-8265-9_80759. URL https://doi.org/10.1007/978-1-4614-8265-9_80759.

[108] B. Saha and D. Srivastava. Data quality: The other face of big data. In *2014 IEEE 30th International Conference on Data Engineering*, 2014. doi: 10.1109/ICDE.2014.6816764.

[109] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017. ISSN 2150-8097. doi: 10.1145/3186728.3164139. URL https://doi.org/10.1145/3186728.3164139.

[110] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 743–754, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. doi: 10.1145/1007568.1007652. URL https://doi.org/10.1145/1007568.1007652.

[111] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB*, 7(12):1059–1070, 2014.

[112] Kai-Uwe Sattler, Stefan Conrad, and Gunter Saake. Adding conflict resolution features to a query language for database federations. *AJIS*, 8(1), 2000. ISSN 1449-8618. doi: 10.3127/ajis.v8i1.262. URL http://journal.acs.org.au/index.php/ajis/article/view/262.

[113] Sheldon Shen. Database relaxation: An approach to query processing in incomplete databases. *Information Processing and Management*, 1988. ISSN 0306-4573. doi: https://doi.org/10.1016/0306-4573(88)90107-0. URL http://www.sciencedirect.com/science/article/pii/0306457388901070.

[114] Arnab Sinha et al. An Overview of Microsoft Academic Service (MAS) and Applications. WWW '15 Companion, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3473-0.

[115] Wee Meng Soon, Hwee Tou Ng, and Chung Yong Lim. A Machine Learning Approach to Coreference Resolution of Noun Phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[116] Michael Stonebraker et al. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.

[117] Dan Suciu, Dan Olteanu, R. Christopher, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011. ISBN 1608456803, 9781608456802.

[118] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1083–1098, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3380581. URL https://doi.org/10.1145/3318464.3380581.

[119] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion

tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, September 2013. ISSN 2150-8097. doi: 10.14778/2556549.2556574. URL https://doi.org/10.14778/2556549.2556574.

[120] M. Tang, R. Y. Tahboub, W. G. Aref, M. J. Atallah, Q. M. Malluhi, M. Ouzzani, and Y. N. Silva. Similarity group-by operators for multi-dimensional relational data. *IEEE Transactions on Knowledge and Data Engineering*, 28(02):510–523, feb 2016. ISSN 1558-2191. doi: 10.1109/TKDE.2015.2480400.

[121] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815410. URL https://doi.org/10.1145/2815400.2815410.

[122] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, page 333–342, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323512. doi: 10.1145/2556195.2556213. URL https://doi.org/10.1145/2556195.2556213.

[123] Niyamat Ullah. Let's analyze our Air that we take, 2019.

[124] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. ISSN 0004-5411. doi: 10.1145/321921.321925. URL https://doi.org/10.1145/321921.321925.

[125] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *ACM J. Exp. Algorithmics*, 15, February 2011. ISSN 1084-6654. doi: 10.1145/1671970.1921702. URL https://doi.org/10.1145/1671970.1921702.

[126] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347808. doi: 10.1145/2960414.2960421. URL https://doi.org/10.1145/2960414.2960421.

[127] Ruben Verborgh and Max De Wilde. *Using OpenRefine*. Packt Publishing, 2013.

[128] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985. ISSN 0098-3500. doi: 10.1145/3147.3165. URL http://doi.acm.org/10.1145/3147.3165.

[129] Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. A sample-and-clean framework for fast and accurate query processing

on dirty data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 469–480, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2610505. URL https://doi.org/10.1145/2588555.2610505.

[130] Reynold Xin. Made sort-based shuffle the default implementation, Spark Issue 3280, 2014.

[131] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. Don't be scared: Use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2463706. URL http://doi.acm.org/10.1145/2463676.2463706.

[132] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.

Styliani-Asimina Giannakopoulou

Curriculum Vitae

✉ *stella.giannakopoulou@epfl.ch*
people.epfl.ch/stella.giannakopoulou
Office BC 242
EPFL/IC/IIF/DIAS

Education

2015-2021 **Ph.D. in Computer Science**, *École Polytechnique Fédérale de Lausanne.*

- Thesis: Holistic, Efficient and Real-time Cleaning of Heterogeneous Data
- Supervisor: Anastasia Ailamaki

2011-2014 **M.Sc. in Advanced Information Systems**, *University of Athens*, Greece.

- GPA: 8.86 out of 10.0
- Master Thesis: A Reasoner for the RCC-5 and RCC-8 Calculi Extended with Constants
- Supervisor: Manolis Koubarakis

2007-2011 **Bachelor of Informatics and Telecommunications**, *University of Athens*, Greece.

- GPA: 8.94 out of 10.0
- Thesis: Storing RDF data and processing SPARQL queries using MapReduce
- Supervisor: Manolis Koubarakis

Honors & Awards

- Teaching Assistant Award, 2020
- Teaching Assistant Award, 2018
- EPFL I&C Fellowship Student, 2015-2016
- University of Athens: ranked 2nd and 3rd respectively for the academic years '07-'08, '08-'09.

Work Experience

2015–now **Research Assistant**, *EPFL*, Switzerland.

- Member of the Data-Intensive Applications and Systems laboratory
- Worked on efficient cleaning of heterogeneous data

2018 **Microsoft Research**, *Redmond*, Seattle.

- Summer Intern at the Data Management, Exploration and Mining group (DMX)
- Mentors: Vivek Narasayya and Surajit Chaudhuri
- Worked on sampling solutions for duplicate detection

2012-2015 **Research Assistant/Developer**, University of Athens, Greece.

- Worked on project SCARE: Scalable query processing and reasoning for linked geospatial data
- Worked on the EU project LEO: Linked Open Earth Observation Data for Precision Farming http://www.linkedeodata.eu/
- Worked on the EU project MELODIES: Exploiting Open Data http://www.melodiesproject.eu/
- Worked on the EU project TELEIOS: Virtual Observatory Infrastructure for Earth Observation Data http://www.earthobservatory.eu/

Publications

- **Stella Giannakopoulou**, Manos Karpathiotakis, Anastasia Ailamaki: Cleaning Denial Constraint Violations through Relaxation. SIGMOD Conference 2020: 805-815
- **Stella Giannakopoulou**, Manos Karpathiotakis, Anastasia Ailamaki: Query-driven Repair of Functional Dependency Violations. ICDE 2020: 1886-1889
- Matthaios Olma, **Stella Giannakopoulou**, Manos Karpathiotakis, Anastasia Ailamaki: Toward Intelligent Query Engines. IEEE Data Eng. Bull. 42(2): 7-18 (2019)

- **Stella Giannakopoulou**, Manos Karpathiotakis, Benjamin Gaidioz, Anastasia Ailamaki: CleanM: An Optimizable Query Language for Unified Scale-Out Data Cleaning. PVLDB 10(11): 1466-1477 (2017)
- **Stella Giannakopoulou**, Charalampos Nikolaou, Manolis Koubarakis: A Reasoner for the RCC-5 and RCC-8 Calculi Extended with Constants. In Proceedings of the 28th AAAI Conference on Artificial Intelligence, AAAI'14, 2014.
- Managing Big, Linked, and Open Earth-Observation Data Using the TELEIOS/LEO software stack, Manolis Koubarakis, Kostis Kyzirakos, Charalampos Nikolaou, George Garbis, Konstantina Bereta, Roi Dogani, **Stella Giannakopoulou**, Panayiotis Smeros, Dimitrianos Savva, George Stamoulis, Giannis Vlachopoulos, Stefan Manegold, Charalampos Kontoes, Themistocles Herekakis, Ioannis Papoutsis, and Dimitrios Michail, IEEE Geoscience and Remote Sensing Magazine, Vol. 4, Issue 3, p. 23-37, September 2016
- C. Nikolaou, K. Kyzirakos, K. Bereta, K. Dogani, **S. Giannakopoulou**, P. Smeros, G. Garbis, M. Koubarakis, D. E. Molina, O. C. Dumitru, G. Schwarz and M. Datcu. Improving knowledge discovery from synthetic aperture radar images using the linked open data cloud and Sextant. In Informal Proceedings of the Image Information Mining Conference: The Sentinels Era (ESA-EUSC-JRC 2014), 2014.
- Big, Linked and Open Data: Applications in the German Aerospace Center, Charalampos Nikolaou, Kostis Kyzirakos, Konstantina Bereta, Kallirroi Dogani, **Stella Giannakopoulou**, Panayiotis Smeros, George Garbis, Manolis Koubarakis, Daniela Espinoza-Molina, Corneliu Octavian Dumitru, Gottfried Schwarz, Mihai Datcu, ESWC (Satellite Events) 2014: 444-449
- M. Koubarakis, K. Kyzirakos, C. Nikolaou, G. Garbis, K. Bereta, P. Smeros, **S. Giannakopoulou**, K. Dogani, M. Karpathiotaki and I. Vlachopoulos. Linked Earth Observation Data:The Projects TELEIOS and LEO. In the Linking Geospatial Data Conference (LGD 2014). Campus London, Shoreditch. 5-6 March 2014
- Linked Open Data in the Earth Observation Domain: the Vision of Project LEO, Manolis Koubarakis, Panayiotis Smeros, Charalampos Nikolaou, George Garbis, Konstantina Bereta, **Stella Giannakopoulou**, Kallirroi Dogani, Maria Karpathiotaki, Ioannis Vlachopoulos, Dimitrianos Savva, George Stamoulis, Kostis Kyzirakos, Stefan Manegold, Bernard Valentin, Heike Bach, Fabian Niggemann, Philipp Klug, Wolfgang Angermair and Stefan Burgstaller, Big Data from Space (BiDS '14). ESA-ESRIN, Frascati, Italy, November 12-14, 2014
- Kostis Kyzirakos, Manos Karpathiotakis, Konstantina Bereta, George Garbis, Charalampos Nikolaou, Panayiotis Smeros, **Stella Giannakopoulou**, Kallirroi Dogani, Manolis Koubarakis: The Spatiotemporal RDF Store Strabon. In Proceedings of the 13th International Conference on Advances in Spatial and Temporal Databases, SSTD '13, pages 496–500, 2013.
- M. Koubarakis, S. Manegold, C. Kontoes, M. Karpathiotakis, K. Kyzirakos, K. Bereta, G. Garbis, C. Nikolaou, I. Papoutsis, T. Herekakis, M. Ivanova, Y. Zhang, H. Pirk, M. Kersten, K. Dogani, **S. Giannakopoulou** and P. Smeros. Real-Time Wildfire Monitoring Using Scientific Database and Linked Data Technologies. In the 16th International Conference on Extending Database Technology (EDBT), March 18-22, 2013. Genoa, Italy.

Teaching

2016-Now **Teaching Assistant**, *EPFL*.

- Programmation I (Fall Semester)
  Instructor: Sam Jamila
- Database Systems (Spring Semester)
  Instructors: Ailamaki Anastasia, Papapetrou Odysseas
- Programmation I (Fall Semester)
  Instructor: Ronan Boulic
- Introduction à la programmation orientée objet (en C++) (Spring Semester)
  Instructor: Jean-Cédric Chappelier

2011-2014 **Teaching Assistant**, University of Athens.

- Artificial Intelligence (Fall semester 2011, 2012, 2013)
  Instructor: Manolis Koubarakis

- Artificial Intelligence II (Spring semester 2012, 2013, 2014)
  Instructor: Manolis Koubarakis

- Knowledge Technologies (Fall semester 2013)
  Instructor: Manolis Koubarakis

Professional Activities

- Reviewer: VLDB, SIGMOD

Languages

English

2008 **Certificate of Proficiency in English**, *University of Michigan*.

French

2004 **Diplôme D' Études en Langue Française (DELF) 1er Degré**.

Greek

Native.

References

Available upon request