University of Potsdam
Hasso Plattner Institute
Information Systems Group

Single-column Data Profiling

Dissertation
zur Erlangung des akademischen Grades
"Doktor der Ingenieurwissenschaften"
(Dr.-Ing.)
in der Wissenschaftsdisziplin
"Informationssysteme"

eingereicht an der
Digital Engineering Fakultät
der Universität Potsdam

von
Hazar Harmouch

Potsdam, den 4. März 2020

This work is licensed under a Creative Commons License:
Attribution 4.0 International.
This does not apply to quoted content from other authors.
To view a copy of this license visit
https://creativecommons.org/licenses/by/4.0/

Reviewers

Professor Dr. Felix Naumann
Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam

Professor Dr. Wolfgang Lehner
Faculty of Computer Science Institute of System Architecture, The Technical University of Dresden

Professor Dr. Ziawasch Abedjan
Faculty of Electrical Engineering and Computer Science, The Technical University of Berlin

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-47455
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-474554

Abstract

The research area of data profiling consists of a large set of methods and processes to examine a given dataset and determine metadata about it. Typically, different data profiling tasks address different kinds of metadata, comprising either various statistics about individual columns (Single-column Analysis) or relationships among them (Dependency Discovery). Among the basic statistics about a column are data type, header, the number of unique values (the column's cardinality), maximum and minimum values, the number of NULL values, and the value distribution. Dependencies involve, for instance, functional dependencies (FDs), inclusion dependencies (INDs), and their approximate versions.

Data profiling has a wide range of conventional use cases, namely data exploration, cleansing, and integration. The produced metadata is also useful for database management and schema reverse engineering. Data profiling has also more novel use cases, such as big data analytics. The generated metadata describes the structure of the data at hand, how to import it, what it is about, and how much of it there is. Thus, data profiling can be considered as an important preparatory task for many data analysis and mining scenarios to assess which data might be useful and to reveal and understand a new dataset's characteristics.

In this thesis, the main focus is on the single-column analysis class of data profiling tasks. We study the impact and the extraction of three of the most important metadata about a column, namely the cardinality, the header, and the number of NULL values.

First, we present a detailed experimental study of twelve cardinality estimation algorithms. We classify the algorithms and analyze their efficiency, scaling far beyond the original experiments and testing theoretical guarantees. Our results highlight their trade-offs and point out the possibility to create a parallel or a distributed version of these algorithms to cope with the growing size of modern datasets.

Then, we present a fully automated, multi-phase system to discover human-understandable, representative, and consistent headers for a target table in cases where headers are missing, meaningless, or unrepresentative for the column values. Our evaluation on Wikipedia tables shows that 60% of the *automatically* discovered schemata are *exact* and *complete*. Considering more schema candidates, top-5 for example, increases this percentage to 72%.

Finally, we formally and experimentally show the *ghost* and *fake* FDs phenomenon caused by FD discovery over datasets with missing values. We propose two efficient scores, probabilistic and likelihood-based, for estimating the genuineness of a discovered FD. Our extensive set of experiments on real-world and semi-synthetic datasets show the effectiveness and efficiency of these scores.

Zusammenfassung

Das Forschungsgebiet Data Profiling besteht aus einer Vielzahl von Methoden und Prozessen, die es erlauben Datensätze zu untersuchen und Metadaten über diese zu ermitteln. Typischerweise erzeugen verschiedene Data-Profiling-Techniken unterschiedliche Arten von Metadaten, die entweder verschiedene Statistiken einzelner Spalten (Single-Column Analysis) oder Beziehungen zwischen diesen (Dependency Discovery) umfassen. Zu den grundlegenden Statistiken einer Spalte gehören unter anderem ihr Datentyp, ihr Name, die Anzahl eindeutiger Werte (Kardinalität der Spalte), Maximal- und Minimalwerte, die Anzahl an Null-Werten sowie ihre Werteverteilung. Im Falle von Abhängigkeiten kann es sich beispielsweise um funktionale Abhängigkeiten (FDs), Inklusionsabhängigkeiten (INDs) sowie deren approximative Varianten handeln.

Data Profiling besitzt vielfältige Anwendungsmöglichkeiten, darunter fallen die Datenexploration, -bereinigung und -integration. Darüber hinaus sind die erzeugten Metadaten sowohl für den Einsatz in Datenbankmanagementsystemen als auch für das Reverse Engineering von Datenbankschemata hilfreich. Weiterhin finden Methoden des Data Profilings immer häufiger Verwendung in neuartigen Anwendungsfällen, wie z.B. der Analyse von Big Data. Dabei beschreiben die generierten Metadaten die Struktur der vorliegenden Daten, wie diese zu importieren sind, von was sie handeln und welchen Umfang sie haben. Somit kann das Profiling von Datenbeständen als eine wichtige, vorbereitende Aufgabe für viele Datenanalyse- und Data-Mining Szenarien angesehen werden. Sie ermöglicht die Beurteilung, welche Daten nützlich sein könnten, und erlaubt es zudem die Eigenschaften eines neuen Datensatzes aufzudecken und zu verstehen.

Der Schwerpunkt dieser Arbeit bildet das Single-Column Profiling. Dabei werden sowohl die Auswirkungen als auch die Extraktion von drei der wichtigsten Metadaten einer Spalte untersucht, nämlich ihrer Kardinalität, ihres Namens und ihrer Anzahl an Null-Werten.

Die vorliegende Arbeit beginnt mit einer detaillierten experimentellen Studie von zwölf Algorithmen zur Kardinalitätsschätzung. Diese Studie klassifiziert die Algorithmen anhand verschiedener Kriterien und analysiert ihre Effizienz. Dabei sind die Experimente im Vergleich zu den Originalpublikationen weitaus umfassender und testen die theoretischen Garantien der untersuchten Algorithmen. Unsere Ergebnisse geben Aufschluss über Abwägungen zwischen den Algorithmen und weisen zudem auf die Möglichkeit einer parallelen

bzw. verteilten Algorithmenversion hin, wodurch die stetig anwachsende Datenmenge moderner Datensätze bewältigt werden könnten.

Anschließend wird ein vollautomatisches, mehrstufiges System vorgestellt, mit dem sich im Falle fehlender, bedeutungsloser oder nicht repräsentativer Kopfzeilen einer Zieltabelle menschenverständliche, repräsentative und konsistente Kopfzeilen ermitteln lassen. Unsere Auswertung auf Wikipedia-Tabellen zeigt, dass 60% der automatisch entdeckten Schemata exakt und vollständig sind. Werden darüber hinaus mehr Schemakandidaten in Betracht gezogen, z.B. die Top-5, erhöht sich dieser Prozentsatz auf 72%.

Schließlich wird das Phänomen der Geist- und Schein-FDs formell und experimentell untersucht, welches bei der Entdeckung von FDs auf Datensätzen mit fehlenden Werten auftreten kann. Um die Echtheit einer entdeckten FD effizient abzuschätzen, schlagen wir sowohl eine probabilistische als auch eine wahrscheinlichkeitsbasierte Bewertungsmethode vor. Die Wirksamkeit und Effizienz beider Bewertungsmethoden zeigt sich in unseren umfangreichen Experimenten mit realen und halbsynthetischen Datensätzen.

Acknowledgements

First, I want to express my appreciation and gratitude to my advisor *Felix Naumann*, who was always a supportive, patient, and caring mentor. This thesis would not have been possible if he didn't believe in me. I am truly thankful for the guidance, the knowledge and the support I received from him.

My gratitude also goes to *Deutscher Akademischer Austauschdienst* (DAAD) for the great opportunity they gave to me at the hardest part of my life. Also, many thanks to *Laure Berti-Équille, Noël Novelli, and Saravanan Thirumuruganathan*, I really enjoined our collaboration and discussions.

Furthermore, I am truly grateful for having wonderful friends in the chair who helped me feel at home. I want to express special thanks go to *Thorsten* with whom I had the pleasure to work with and learn from and to *Michael* who supported and encouraged me during the final stage of my PhD.

Finally and most importantly, I want to thank my family for their endless support and love which gave me the confidence to follow my dream. Without them, I would not be who I am.

Contents

1 Metadata: A Mediator between People, Systems and Data **1**

1.1 Data profiling . . . 3
1.2 Single-column data profiling . . . 4
1.3 Structure and contributions . . . 8

2 Cardinality Estimation **11**

2.1 Cardinality: The zeroth-frequency moment . . . 12
2.2 Classification of general approaches and algorithms . . . 14
2.3 Review of twelve cardinality estimation algorithms . . . 18
2.4 Comparative experiments . . . 27
2.5 Summary . . . 38

3 Discovering Missing Column Headers **39**

3.1 Missing schema: Dark data . . . 40
3.2 Related work . . . 44
3.3 Similarity search . . . 45
3.4 Topic coherence . . . 50
3.5 Missing schema discovery . . . 52
3.6 Experiments . . . 58
3.7 Summary . . . 69

4 The Impact of Missing Values on FD Discovery **71**

4.1 FDs and incomplete data: Trust . . . 72
4.2 Related work . . . 76
4.3 Genuine, Ghost, and Fake FDs . . . 77
4.4 Identifying genuine FDs . . . 81
4.5 Probabilistic FD genuineness . . . 81
4.6 Likelihood-based FD genuineness . . . 85

4.7 Experiments 86
4.8 Summary 97

5 Conclusion and Outlook **99**

References **103**

Chapter 1

Metadata: A Mediator between People, Systems and Data

> “Metadata liberates us, liberates knowledge”
>
> Dr. David Weinberger

The main challenge facing data scientists today is finding (and cleaning) the right datasets for a given data science scenario [Miller, 2018]. As reported in [Press, 2016], data scientists spend 19% of their time on selecting datasets and another 60% on cleaning them amounting to a total of 89% of the entire task. In the big data universe, the volume and variety of data sources as well as their heterogeneous formats exacerbate this problem. Stock exchange, social media, online retail, search engines as well as sensor data of jets, satellites, and weather stations are all examples of such data sources that generate a tremendous amount of data per second. In addition, challenging and valuable data is available in *data lakes*. These repositories contain structured, semi-structured, or unstructured data in its raw natural state. Two examples of data lakes are open data and web tables [Miller, 2018].

The ability to understand, explore, and query such data is limited and therefore hardly usable in any later applications. *“If we just have a bunch of data sets in a repository, it is unlikely anyone will ever be able to find, let alone reuse, any of this data. with adequate metadata, there is some hope, but even so, challenges will remain...”* [Agrawal et al., 2012]. Metadata is as valuable as the data itself, because it reveals yet unknown properties of the data. Thus, it gives us the opportunity to discover the good, bad, and ugly about the data. In other words, it acts as a mediator between the data, data scientists, researchers, IT professionals, and any downstream systems.

Generally, metadata is a crucial asset in data exploration [Kruse et al., 2016], cleaning [Rekatsinas et al., 2017], and integration [Wang et al., 2009]. It is also essential in other data management tasks, such as schema engineering [Papenbrock and Naumann, 2017] and query optimization [Liu et al., 2016].

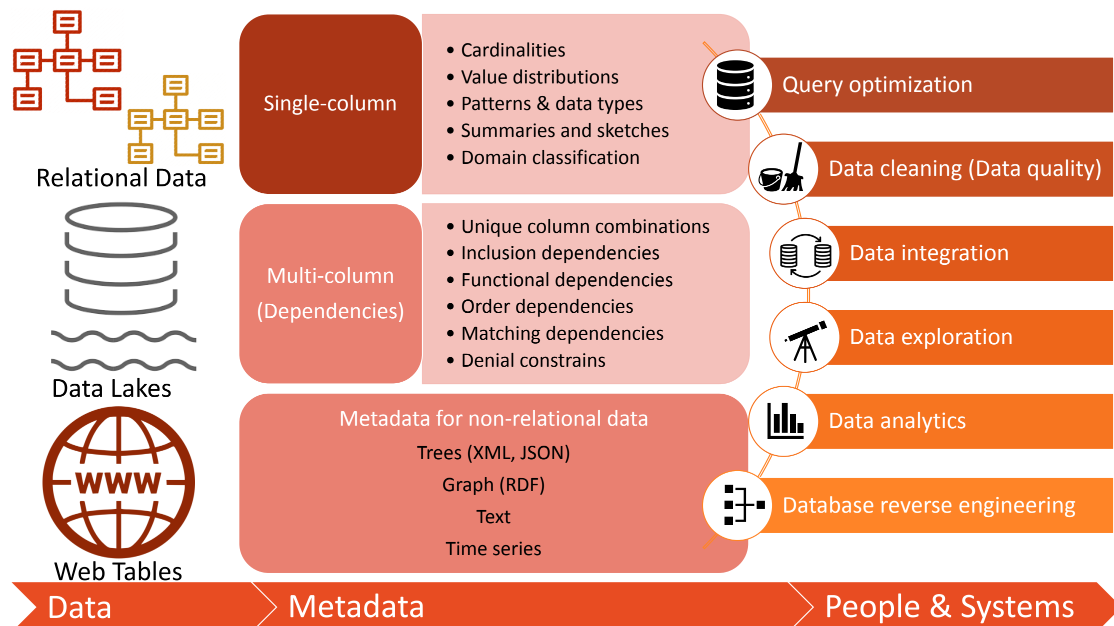


Figure 1.1: An overview of typical metadata types produced by data profiling, and their use cases.

Metadata can be extracted from many data formats, such as relations (tables), trees (XML,YAML, and JSON), graph (RDF), and free text. In the context of relational data, the types of metadata can be categorized into either single-column or multi-column metadata based on their scope [Naumann, 2014]. Whereas *single-column* metadata provides a number of statistical information about individual columns, for example, cardinalities, data types, and value distributions, *multi-column* metadata expresses relationships and correlations between a group of columns in one or more tables, such as foreign keys, inclusion dependencies, and functional dependencies.

However, such metadata is in many cases missing, either because of issues encountered during data transformation, data transportation, or it never existed. Even datasets with rich metadata may be growing too fast to keep metadata up to date. The field of *data profiling* focuses on efficient, salable, and automatic extraction of both single-column and multi-column metadata [Abedjan et al., 2018]. Data profiling has been applied by data professionals to understand the content, the structure, and the quality of datasets and and assess their suitability for a specific data science scenario. An overview of data profiling use cases and metadata types is shown in Figure 1.1. The individual parts of this figure are explained throughout this chapter.

This thesis focuses on data profiling tasks for extracting metadata about single columns in relational datasets, specifically their cardinality, their number of NULL values, and their header. In this introductory chapter, we first give an overview of data profiling. Afterward, we focus on single-column profiling tasks and discuss their importance and use cases. Finally, We introduce our concrete contributions and the structure of this thesis.

1.1 Data profiling

Data profiling is *"the activity of creating small but informative summaries of a database"* [Johnson, 2009]. Data profiling is also defined as a collection of tasks to systematically inspect a given dataset and automatically extract metadata about it [Abedjan et al., 2018]. In general, a data profiling task expects only the data itself as an input and generates as its output a set of metadata called the *data profile*.

With the focus on relational data, data profiling tasks are classified into two classes based on metadata type:

- **Single-column profiling** encompasses the tasks that result in single-column metadata, such as cardinality (Chapter 2), data type, header/semantic type (Chapter 3), or the number of NULL values (Chapter 4).
- **Multi-column profiling** consists of the set of profiling tasks that generate multi-column metadata, such as functional dependencies [Papenbrock et al., 2015b], inclusion dependencies [Dürsch et al., 2019] or denial constrains [Pena et al., 2019].

Regardless of their class, data profiling tasks are either *exact* or *approximate*. As an example, functional dependency discovery algorithms can either generate exact functional dependencies or their approximate form. As such, the requirement for an FD to hold can be relaxed, resulting in the detection of *approximate* (a.k.a. partial) or *relaxed* FDs. While approximate FDs are violated by some fraction of the tuples, relaxed FDs allow similar attribute values and do not require them to be identical when checking for violations [Caruccio et al., 2016].

The importance of data profiling is a natural result of its implications in many use cases. *Query optimization* is one of the most prominent use cases of data profiling. For instance, the generated metadata can directly support the process of estimating the cost of a query plan. Data profiling techniques are powerful tools to detect patterns, data errors, and inconsistencies; therefore, they play a significant role in *data cleaning*. In other words, data profiling assesses *data quality* by quantifying the fraction of data that violates the previously specified constraints.

The metadata generated by data profiling techniques helps data professionals to understand and *explore* a dataset. Moreover, such an informative summary of a *"big" dataset* is necessary to avoid costs arising from the use of large amounts of data without any prior assessment of its content and quality. When there is more than one data source, data profiling reveals the commonalities and the differences between these sources and gives hints to support *data integration* efforts.

A considerable number of contributions have been made to the field of data profiling by both research and industry. We refer the reader to [Abedjan et al., 2018] for an overview. There are also several commercial data profiling tools, such as Trifacta[1],

[1] `https://www.trifacta.com/`

OpenRefine[2], IBM InfoSphere[3], and SAP Information Steward[4]. Apart from these industrial solutions, research also produced tools like Metanome [Papenbrock et al., 2015a], RuleMiner [Chu et al., 2014], and ProLOD++ [Abedjan et al., 2014].

Nevertheless, there are still challenges and open research questions in the field of data profiling, such as how to deal with non-relational, heterogeneous, incremental, and streaming data scenarios. Another dimension to enhance data profiling methods is to improve them so that they remain effective even when faced with data quality problems, such as missing or dirty data. In this thesis, we address the problem of estimating column cardinalities from big data and data streams. We investigate the effect of missing/dirty data on functional dependency discovery and propose a solution to recommend missing headers for columns in a table to resolve this data quality issue and support any further data-driven application.

1.2 Single-column data profiling

Single-column data profiling tasks represent the main focus of this thesis. As shown in Table 1.1, these tasks can be grouped into three categories: cardinalities, value distributions, and data patterns, types, and domains [Abedjan et al., 2018]. The basic statistics collected by DBMS are a subset of single-column metadata [Mannino et al., 1988].

Given a dataset, single-column profiling provides data scientists with a set of potent tools to gain a basic understanding of this dataset. In the rest of this section, we discuss these three categories of single-column profiling tasks. For further reading about other single-profiling tasks, we refer to the following books [Loshin, 2010; Maydanchik, 2007].

1.2.1 Cardinalities (counts)

Different counts can describe the content of an individual column in a table, such as its size, cardinality, and the number of NULL values. Column *size* refers to the number of values within this column, whereas column *cardinality* is the number of distinct values. Put differently, the size of each column in a table is equal to the number of rows or entities represented in this table. Once the cardinality of a column is equal to its size, it is a key candidate. The cardinality and the number of NULL values are the focus of Chapter 2 and Chapter 4 of this thesis, respectively. More details about their use cases and importance are discussed in the dedicated chapters.

Two of the data quality dimensions are based on these counts, namely *uniqueness* and *completeness* [Loshin, 2010]. The completeness dimension implies the absence of missing values. It can be assessed by the ratio of NULL values to column size. In the same manner, the ratio of distinct values to column size can be used to measure the uniqueness dimension. Data uniqueness is characterized by the existence of each value only once within the column.

[2] `https://openrefine.org/`
[3] `https://www.ibm.com/analytics/information-server`
[4] `https://www.sap.com/germany/products/data-profiling-steward.html`

Table 1.1: Overview of single-column profiling [Abedjan et al., 2018] (Tasks in **bold** are the focus of this thesis).

Category	Task	Task Description
Cardinalities	num-rows	Number of rows
	NULL values	Number or percentage of NULL values
	distinct	Number of distinct values
	uniqueness	Number of distinct values divided by number of rows
Value Distributions	histogram	Frequency histograms (equi-width, equi-depth, etc.)
	extremes	Minimum and maximum values in a numeric column
	constancy	Frequency of most frequent value divided by number of rows
	quartiles	Three points that divide (numeric) values into four equal groups
	first digit	Distribution of first digit in numeric values; to check Benford's law [Benford, 1938]
Data Types, Patterns, and Domains	basic type	Numeric, alphanumeric, date, time, etc.
	data type	DBMS-specific data type (varchar, timestamp, etc.)
	lengths	Minimum, maximum, median, and average lengths of values within a column
	size	Maximum number of digits in numeric values
	decimals	Maximum number of decimals in numeric values
	patterns	Histogram of value patterns (Aa9...)
	data class	Generic semantic data type, such as code, indicator, text, date/time, quantity, identifier
	domain	Semantic domain, such as credit card, first name, city, phenotype

1.2.2 Value distributions

A more complex single-column analysis of numerical columns reveals some statistical properties of the content of the column at hand. This analysis generates metadata including, but is not limited to, extremes, quartiles, histogram, and the first digit distribution. We now briefly discuss each of these metadata types.

Extremes and quartiles

Tukey [1977] motivated a five-numbers summary of (extremes and quartiles): the minimum and maximum values, the median, and the lower and upper quartiles. The *lower quartile* is the middle number between the minimum and the median. The *upper quartile* is the middle number between the median and the maximum value. The distance between the upper and lower quartiles serves as a dispersion measure for the dataset. This distance, known as the *interquartile range*, can be used to detect outliers by using box-and-whisker plots [Dekking et al., 2005].

Extremes and quartiles are part of the *order statistics* of a dataset [David and Nagaraja, 2004]. The kth smallest value of an unordered list (e.g., a column or a dataset) represents the kth order statistics. All order statistics regarding a specific column can

be efficiently and exactly extracted using the QuickSelect algorithm: a linear quicksort-based algorithm [Blum et al., 1973; Hoare, 1961]. However, the emergence of data streams and the need for real-time processing of high-volume and velocity data motivated further research efforts to find an approximation of order statistics. Ma et al. [2013] introduced 1-unit and 2-units memory frugal algorithms for estimating order statistics of a data streams. Recently published work by Dunning and Ertl [2019] presents a set of streaming algorithms to accurately compute order statistics, especially when the data is expected to follow a skewed distribution.

Histograms

Another widely used type of graphical summary of datasets are *histograms* [Dekking et al., 2005]. A histogram is a bar graph representation of the frequency distribution of a column. Such graphical summaries are easily interpretable by data scientists; thus, histograms are robust tools for data exploration. There are different flavors of basic histograms, depending on the method of grouping data values into buckets, such as equi-depth, equi-width, and equi-hight histograms [Ioannidis, 2003]. More complex and accurate histograms have been designed with different methods to choose a bucket or different statistics to store [Cormode et al., 2006].

Most modern DBMSs incorporate histograms in their query optimizers to improve the query plan cost estimation using the actual value distribution within the involved columns. The default value for a column can be determined by examining the value distribution of this column [Hua and Pei, 2007]. A histogram can be derived from a dataset by applying a recursive dynamic-programming algorithm [Jagadish et al., 1998]. In practice, this algorithm is not scalable enough; hence, several heuristics have been employed, such as maxdiffare [Poosala et al., 1996], STHoles [Bruno et al., 2001], lattice histograms [Karras and Mamoulis, 2008], and sample boundaries [Halim et al., 2009].

Approximating the value frequencies in a data stream is an active research topic. Several data structures have been designed to keep track of values frequencies by using bounded storage, such as counting bloom filters [Fan et al., 2000] or count-min sketchs [Cormode and Muthukrishnan, 2005]. For an overview of sketches that deal with the value distribution of a dataset, we refer to [Cormode et al., 2011].

Distribution of first digit in numeric values

Benford [1938] observed an interesting statistical phenomenon regarding the probability distribution of the *first digit* in numerical datasets from 20 real-world domains. Benford's law, also known as the first digit law, states that the first digit d ($d \in 1, ..., 9$) of a set of numbers occurs with a probability of $p(d) = \log_{10} 1 + \frac{1}{d}$. Accordingly, in any numerical set that obeys this law, the digit 1 is the leading digit with probability 0.3, i.e., it appears as the first digit about 30% of the time. This observation has been exploited to detect fraudulent behaviour and anomalies in numerical columns that obey this law.

1.2.3 Data types, patterns, and domains

Beyond simple counts and statistical properties of the content of a column, single-column profiling tasks also reveal additional high-level metadata about the format and the se-

mantic of a column. Profiling tasks of this category convey metadata of varying semantic granularity levels. So, they are ordered in Table 1.1 by increasing semantic richness as well as increasing discovery difficulty. We now briefly discuss each subcategory.

Data type

A basic form of such metadata is the *basic type* (e.g., numeric vs. alphabetic and date) and the *DBMS-specific type* (e.g., varchar vs. timestamp) of a column. Each data type has some unique characteristics that make it easily distinguishable. For example, profiling tools check for the absence of numbers, the presence of numbers with no-decimals, and value ranges to detect alphabetic columns, integers and dates, respectively. Nevertheless, it is not trivial to detect some data types, because they are a generalization of each other, such as integer, float, and double. Profiling tools usually aim for the more specific type. Extracting columns data types directly supports any schema reverse-engineering effort.

Patterns

Some profiling tasks identify *patterns* in the data values in the form of regular expressions (regexes) or a formal language. Raman and Hellerstein [2001] use the minimum description length (MDL) to infer suitable patterns that represent the values of a column. The information extraction system proposed by Li et al. [2008] creates regexes by training on negative and positive examples. Fernau [2009] formalized the problem of learning regexes from data, while Bartoli et al. [2016, 2017] introduced a system for generating regexes that is based on genetic programming and uses multi-criteria optimization search to explore the vast solution space.

The main challenge to any pattern inference algorithm is the level of granularity of the proposed pattern, i.e., how general or specific the pattern should be. For example, $(.*)$, $(\backslash d\{4\})$ and $((19|20) \backslash d\{2\})$ are regexes that can match a year column in the 20th or 21st centuries. Unfortunately, the current methods to extract regexes do not scale well for large datasets. For instance, learning the regexes from 500 examples needs more than 40 minutes [Bartoli et al., 2014]. To overcome this problem, the XSystem efficiently learns a simpler syntactic pattern that represents a column at hand, and can then be used to identify outliers or assign a semantic label to this column [Ilyas et al., 2018].

As a use case, any value violating the inferred patterns is recognized as an outlier, i.e., an error candidate. To automatically detect such incompatible values within a column, the Auto-Detect technique maps each column values to generalized languages, each of them is sensitive to different types of errors [Huang and He, 2018]. Qahtan et al. modeled partial column values as regex-like patterns and combined them with integrity constraints as part of their solution for data cleaning [Qahtan et al., 2019].

Domains

Single-column profiling tasks are also used to assign a *semantic data type or a domain* to a column. With the need to interpret the semantics of the data, the difficulty of the problem increases. In Chapter 3, we propose an automated system that is able to assign a meaningful header to a header-less column. We also discuss several approaches of assigning a semantic class from a knowledge base to a column and how well such classes are suited as headers.

1.3 Structure and contributions

This thesis focuses on the metadata of a single column in relational tables, and thus on single-column profiling. We present the importance, utilization and discovery of cardinality and headers as two important types of single-column metadata. In addition, we analyze the influence of another metadata, namely the number of NULL values in a column, on dependency discovery.

Particularly, we make the following contributions:

(1) An experimental survey of cardinality estimation (*Chapter 2*)

We discuss several broad approaches that are used to estimate the cardinality of a multiset and their limitations . Then, we present three fine-grained classifications of cardinality estimation algorithms and provide a novel classification that distinguishes the core method of these algorithms. To guarantee a unified test environment when comparing the twelve algorithms, we implement them and extend the Metanome data profiling framework to support these algorithms. Our implementation of the algorithms as well as Metanome are open source. We then evaluate the twelve algorithms' accuracy, runtime, and memory consumption using synthetic and real-world datasets. In addition, we evaluate the Guaranteed-Error Estimator as an example of a sampling-based cardinality estimator. Our results show that different algorithms excel in different categories, and we highlight their trade-offs. We also point out the possibility to create a parallel or a distributed version of these algorithms to cope with the growing size of modern datasets.

This chapter is based on our published work in [Harmouch and Naumann, 2017] which has been marked as reproducible by the pVLDB Reproducibility Committee [5].

(2) Automatic end-to-end schema discovery system (*Chapter 3*)

A schema discovery system that automatically extracts meaningful column headers for given relational tables from a corpus of (web) tables. The system runs an efficient similarity search algorithm optimized for matching web table columns. Furthermore, we choose an unbiased similarity measure for web table columns that efficiently matches similar value sets regardless of their length. The system has a context-aware column header assessment: A schema evaluation approach that finds the overall optimal label composition for a schema given multiple sets of column candidate headers. Finally, we provide a systematic, empirical evaluation of the effectiveness of our schema discovery system on hundreds of real-world web tables. A paper based on the content of this chapter is under submission.

(3) Efficient methods to estimate an FD genuineness (*Chapter 4*)

We formally and experimentally show the phenomenon caused by missing values over FD discovery under various NULL semantics and imputation strategies. A *genuine* FD is an FD that would be valid if the dataset contained no missing values and no other errors. We propose two efficient approaches to approximate the genuineness score of discovered FDs: Sampling-based and Likelihood-based (PerValue and PerTuple). We perform an

[5] https://vldb-repro.com/

extensive set of experiments of our methods on real-world and semi-synthetic datasets that show the effectiveness and efficiency of our suggested scores.

The contribution of this chapter was a joint effort of Berti-Équille, Harmouch, Naumann, Novelli, and Thirumuruganathan [Berti-Equille et al., 2018]. Novelli and Thirumuruganathan contributed the efficient exact probabilistic genuineness score algorithm and the sampling-based approximation of this score. Harmouch and Berti-Équille investigated and formalized the phenomena caused by missing values over FD discovery, contributed the PerValue and PerTuple genuineness scores and the case study on the Sensor dataset. Harmouch performed all experiments.

We conclude this thesis in Chapter 5 by summing up our results, the potential implications, and discussing open research questions for future work on single-column data profiling.

Chapter 2

Cardinality Estimation

Data profiling comprises many both basic and complex tasks to analyze a dataset at hand and extract metadata. Among the most important types of metadata is the number of distinct values in a column, also known as the *cardinality* or the zeroth-frequency moment. Finding multiset's cardinality is an active research area, because of its ever-growing number of applications in a wide range of computer science domains.

Cardinality estimation is a fundamental task in database query processing and optimization [Youssefi and Wong, 1979]. The query optimizer picks the optimal query plan based on a cost model that uses the cardinality of the attributes in the queried database tables. Erroneous cardinality estimation translates into slow queries, and thus unpredictable performance.

In network security monitoring, DDoS attacks can be identified by analyzing traffic flows and computing the number of distinct flows per target IP [Estan et al., 2003]. In search engines and online data mining, the number of distinct users have seen an advertisement, searched for a word in a search engine, or clicked on a specific URL is an important metadata [Heule et al., 2013; Metwally et al., 2008].

Moreover, the number of distinct values is used in connectivity analysis of Internet topology to find the distance between a pair of nodes on the Internet graph [Palmer et al., 2001]. Similarly, the size of a social network can be estimated by the number of distinct users who have a specific relationship distance [Backstrom et al., 2012].

The aim of this chapter is to review the literature of cardinality estimation and to present a detailed experimental study of twelve algorithms, scaling far beyond the original experiments. This chapter is based on our published work in [Harmouch and Naumann, 2017] which has been marked as reproducible by pVLDB Reproducibility Committee [1].

In detail, our contributions are the following:

1. We discuss several broad approaches that are used to estimate the cardinality of a multiset and the limitation of each one.

[1] https://vldb-repro.com/

2. We present three fine-grained classifications of cardinality estimation algorithms and provide a new novel classification that distinguishes the core method of these algorithms.

3. To guarantee a unified test environment when comparing the twelve algorithms, we re-implemented them and extended the Metanome data profiling framework to support these algorithms.

4. We evaluate the twelve algorithms' accuracy, runtime, and memory consumption using synthetic and real-world datasets. Our results show that different algorithms excel in different in categories, and we highlight their trade-offs.

5. we evaluate Guaranteed-Error estimator as an example of a sampling-based cardinality estimator.

6. We point out the possibility to create a parallel or a distributed version of these algorithms to cope with the growing size of modern datasets.

The rest of this chapter is organized as follows. We first formally define the problem of finding the cardinality of a dataset in Section 2.1. Then, we present general approaches used in literature to solve this problem, and discuss several classifications of concrete algorithms to estimate the cardinality of a dataset in Section 2.2. In Section 2.3, we present, discuss, and compare twelve well-known algorithms – we describe their main idea, error-guarantees, advantages, and disadvantages. Section 2.4 presents our comprehensive set of comparative experiments using both synthetic and real-world data, and reports the results of the empirical evaluation. Finally, we conclude in Section 2.5.

2.1 Cardinality: The zeroth-frequency moment

The problem of finding the number of distinct values of a multiset is polyonymous: In statistics, it is known as the problem of estimating the number of species in a population. It is also know as the cardinality of a column or the "COUNT DISTINCT" in database literature. Furthermore, the number of distinct values in a multiset is referred to as the zeroth-frequency moment by Alon, Marias, and Szegedy, who introduced the frequency moments of a multiset [Alon et al., 1996]:

Definition 2.1. *Consider a multiset $E = (e_1, e_2, \ldots, e_n)$ of n items where each e_i is a member of a universe of N possible values and multiple items may have the same value. Let $m_i = |\{j : e_j = i\}|$ denote the number of occurrences of $i \in N$ in the multiset E. The* frequency moments *F_k for each $k \geq 0$ are*

$$F_k = \sum_{i=1}^{n} m_i^k$$

The number of distinct values in E, called the zeroth-frequency moment F_0 of the multiset E, is the number of elements from universe N appearing at least once in E. For

most applications NULL values are discarded. The size of E is the first-frequency moment F_1. The second-frequency moment F_2, known as *Gini index*, measures the homogeneity or the skewness of E. In a database context, F_2 represents the self join size. Generally, the frequency moments form a set of critical demographic statistics about the data that is needed in many applications such as DBMS, data partitioning in distributed systems, and event detection and monitoring in data streams. The algorithms for finding F_0 are the main topic of this chapter.

Without doubt, given a memory size linear to the size of the dataset makes finding the cardinality an easy task: sort, then count. Nevertheless, such memory need is too much for some applications. Therefore, many algorithms to approximate the cardinality of a dataset have been developed in a manner reducing resource/memory consumption. The other cost-dimension to consider is that of I/O. However, it has been shown that approaches that save cost by sampling cannot guarantee any reasonable degree of accuracy (we also validate this statement in Section 2.4.5). Thus, research has focused on reducing memory consumption and assumes to read all data only once.

The cardinality F_0 has a wide variety of applications. Each application has its special requirements for designing an algorithm determining F_0. Some of these applications require a very accurate estimation of F_0. However, others accept a less accurate estimation. To give an illustration, the number of distinct visitors of a website influences the price of showing advertisements. So allowing only a small error in measuring F_0 is important. In comparison, a more rough estimation of the number of distinct connections is enough to detect a potential denial of service attack.

To address these differing needs, some applications focus on high accuracy but have a high memory consumption and runtime. Others do the opposite and accept lower accuracy and can better limit memory and runtime. As a result, the key requirements for F_0 estimation algorithms for a specific application can be specified by trading off among accuracy, memory consumption, and runtime. This chapter presents many well-known algorithms with which the cardinality can be estimated in big datasets using small additional storage and a small number of operations per element. Our results serve as a guide to choose a suitable algorithm for a given use-case.

To experimentally quantify the accuracy of such algorithms, we check how close the estimation is to the true cardinality. There are many error metrics to evaluate the accuracy of an estimation algorithm; the most popular ones are:

Definition 2.2. *The* standard error *of an estimation $\hat{F}_0$ is the standard deviation of F_0 divided by F_0:*

$$E_{standard}(\hat{F}_0) = \frac{\sigma_{\hat{F}_0}(F_0)}{F_0} \tag{2.1}$$

Definition 2.3. *The* relative error *of an estimation $\hat{F}_0$ is:*

$$E_{relative}(\hat{F}_0) = \frac{|\hat{F}_0 - F_0|}{F_0} \tag{2.2}$$

Another frequently used accuracy metric addresses the strength of the algorithm guarantee as follows:

Definition 2.4. *The* (ε, δ) approximation scheme *of an estimation* $\hat{F_0}$ *means that the estimator guarantees a relative error of* ε *with probability* $\geq 1 - \delta$ *where* $\varepsilon, \delta < 1$.

Besides the accuracy, we also need to quantify the memory consumption of the cardinality estimation algorithm. The data structure maintained by the estimation algorithm in main memory is called *synopsis*. An estimation algorithm should find an estimate $\hat{F_0}$ of the dataset cardinality close to the true F_0 as a function of the synopsis size. To exactly determine F_0 by sorting, the synopsis size needs to be essentially proportional to the size of the multiset.

However, multisets today tend to be too big to fit in main memory of one machine or in the allotted memory for the profiling process. Consequently, the synopsis of the estimation algorithm can be seen from two different perspectives. First, the synopsis is only a temporary data structure used to estimate F_0 in static scenarios, i.e., the whole dataset is stored on disk. Second, the synopsis is a replacement or a compact representation of the real data in scenarios where we cannot store the dataset, such as in streaming applications.

To sum up, the goal of this experimental survey is to present and analyze the efficiency of a wide range of known algorithms for estimating F_0, the number of distinct elements/cardinality of a multiset. We also take into account the application requirements determined by the three factors: accuracy, memory consumption, and runtime.

2.2 Classification of general approaches and algorithms

Here we discuss several broad approaches that are used to estimate the cardinality of a multiset and the limitation of each one. We also present several classifications of the larger set of algorithms presented in this chapter to find an approximation of the number of distinct values. Cardinality can be estimated using the following broad approaches:

Sorting

For decades, the method of determining F_0 was through *sorting* to eliminate duplicates [Whang et al., 1990]. However, sorting is an expensive operation that requires a synopsis size at least as large as the dataset itself. Modern sorting methods have less memory consumption. Still, sorting is an impractical approach especially for the current big datasets.

Bitmap

A trivial method to determine F_0 exactly is by using a *bitmap* of size N, the size of the universe, as the synopsis. The bitmap is initialized to 0s. Then, we scan the dataset item-wise and set the bit i to 1 whenever an item with the i-th value of the universe is observed. After a single scan of the dataset, F_0 is the number of 1s in the bitmap. The synopsis size is a function of the universe size N, which is potentially much larger than

the size of the dataset itself. Thus, this approach is infeasible, but bitmaps in general do play a role in other approaches.

While the methods above are exact, they are also expensive in both size and runtime. If we relax the need for an exact solution, other approaches are available.

Hashing

A straightforward approach to obtain an estimation of F_0 and scale-down the synopsis size is *hashing*. Hashing eliminates duplicates without sorting and requires only one pass over the dataset to build a hash table. However, a simple application of hashing can be worse than sorting in terms of memory consumption. The more the hash collisions in the hash table, the higher the approximation error. Thus, to accurately capture datasets with high column cardinalities, the hash table would need to be too large to fit in normal main memory.

Bitmap of hash values

Instead of storing the hash table, the bitmap approach is used to keep track of the hashed values of the dataset items. Each item i is mapped to a bit $h(i)$ in a bitmap of size N. Hash collisions remain a source of estimation error. A Bloom filter with several groups of hash functions can be used to reduce the hash collision effect on the estimation quality [Kumar et al., 2006]. However, the main problem of this approach is that it requires prior knowledge of the maximum expected cardinality to choose a good bitmap size and hash range.

Sampling

Another common general approach is *sampling* to reduce the synopsis size. Various studies used this approach for cardinality estimation [Flajolet, 1990; Gibbons, 2001; Haas et al., 1995]. Obviously, F_0 is difficult to estimate from a sample of the dataset. Charikar et al. [2000] presented negative results for estimating F_0 from a sample of a large table: for every estimator based on a small sample, there is a dataset where the ratio between the cardinality estimate and the exact cardinality is arbitrarily large. I.e., if the estimator does not examine a large fraction of the input data, there is no guarantee of low error across all input distributions. These results match the results obtained by Haas and Stokes [1998]; Haas et al. [1995]. There, Haas et al. highlighted that to bound the estimation error within a small constant, almost all the dataset needs to be sampled. Therefore, we can admit that any approach based on sampling is unable to provide a good guaranteed error and we need to read the entire dataset to determine an accurate estimation as we show in the experimental section.

Observations in hash values

Another approach relies on making *observations* on the hashed values of the input multiset elements to reduce the size of synopses, such as the length of a particular prefix in the binary representation of the hashed values. The observations are linked only to cardinality and are independent of both replication and order of the items in the dataset. These observations are then used to infer an estimation $\hat{F}_0$ of the dataset cardinality. Most of the algorithms presented in this survey follow this approach to estimate F_0.

Table 2.1: Classifications of algorithms studied in this survey

Algorithm	**Observables** [Flajolet et al., 2008]	**Intuition** [Metwally et al., 2008]	**Core method** (Sec. 2.3)
FM [Flajolet and Martin, 1985]	Bit-pattern	Logarithmic hashing	Count trailing `1s`
PCSA [Flajolet and Martin, 1985]	Bit-pattern	Logarithmic hashing	Count trailing `1s`
AMS [Alon et al., 1996]	Bit-pattern	Logarithmic hashing	Count leading `0s`
BJKST [Bar-Yossef et al., 2002a]	Order statistics	Bucket-based	Count leading `0s`
LogLog [Durand and Flajolet, 2003]	Bit-pattern	Logarithmic hashing	Count leading `0s`
SuperLogLog [Durand and Flajolet, 2003]	Bit-pattern	Logarithmic hashing	Count leading `0s`
HyperLogLog [Flajolet et al., 2008]	Bit-pattern (order statistics)	Logarithmic hashing	Count leading `0s`
HyperLogLog++ [Heule et al., 2013]	Bit-pattern	Logarithmic hashing	Count leading `0s`
MinCount [Giroire, 2009]	Order statistics	Interval-based	k-th min. value
AKMV [Beyer et al., 2007]	Order statistics	Interval-based	k-th min. value
LC [Whang et al., 1990]	No observable	Bucket-based	Linear synopses
BF [Papapetrou et al., 2010]	No observable	Bucket-based	Linear synopses

The backbone of most modern cardinality estimation algorithms is the work of Flajolet and Martin in the mid-1980's [Flajolet and Martin, 1985]. For that reason, Gibbons in his survey for the literature on distinct-values estimation has a separate family of algorithms named Flajolet and Martin‘s Algorithms [Gibbons, 2007]. Under this family, he classified FM & PCSA [Flajolet and Martin, 1985], AMS [Alon et al., 1996], and LogLog & SuperLogLog [Durand and Flajolet, 2003].

To understand the similarity among the larger set of algorithms presented in this survey, we discuss two more fine-grained classifications of cardinality estimation algorithms. In addition, we provide a new classification that distinguishes the core method of the algorithms. Table 2.1 gives a summary of these algorithms and their class according to each classification.

The first classification is by Flajolet et al. [2008]. The authors classified algorithms in two categories corresponding to the type of observations *bit-pattern observables* and *order statistic observables*. In the first category, the hash values are seen as bit-strings. The algorithms are based on the occurrence of particular bit patterns at the binary string representation of the dataset values. On the other hand, the order statistic observable algorithms consider the hash values as real numbers. The estimation is based on the order statistics rather than on bit patterns in binary representations. The order statistic of rank k is the k-th-smallest value in the dataset, which is not sensitive to the distribution

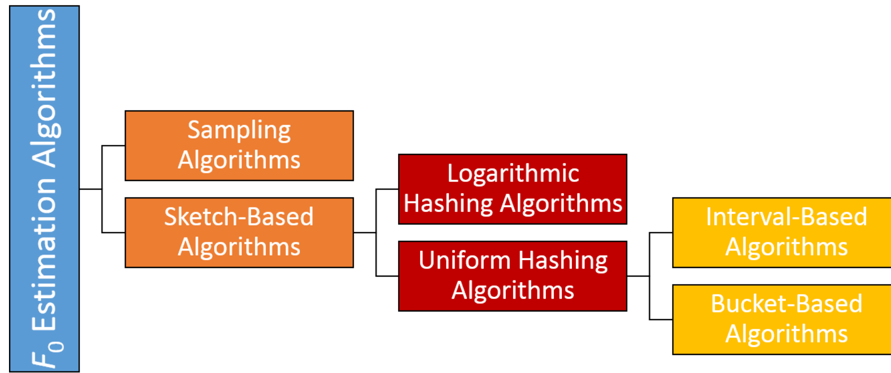


Figure 2.1: Classification of F_0 estimation algorithms [Metwally et al., 2008].

of repeated values. So, the minimum of the hashed values is a good observable. The hash function distributes the hash values uniformly in the interval [0-1]. The minimum of n uniform random variables taking their values in [0-1] is an estimate of $1/(n+1)$. So we are able to retrieve an approximation of n from this value. All the algorithms in our survey belong to the first category except BJKST, MinCount, and AKMV, which are associated with the second category. LC and BF do not use any observable.

The second classification is a high-level classification given by Metwally et al. [2008] (Figure 2.1). The authors distributed the algorithms into two broad categories: *Sampling algorithms* and *sketch-based algorithms*. The first category contains the algorithms that take advantage of not scanning the entire dataset, but estimate the cardinality by sampling (discussed in the previous section). Algorithms of the second category scan the entire dataset once, hash the items, and create a sketch. The sketch, also called synopsis, is queried later on to estimate the cardinality.

Metwally et al. further classified the sketch-based algorithms according to their hashing probabilities into *logarithmic hashing* and *uniform hashing* algorithms. The former keeps track of the most uncommon element observed so far, using a bitmap and a hash function. The hash function maps each element to a bit in the bitmap with a hashing probability that decreases exponentially as the bit significance increases. FM, PCSA, AMS, LogLog, SuperLogLog, HyperLogLog, and HyperLogLog++ are in this category of the sketch-based algorithms. The insight of the latter class, uniform hashing, is to employ a uniform hash function to hash the entire dataset into an interval or a set of buckets. Thus, this class comprises two classes: *Interval-based* algorithms and *Bucket-based* algorithms. F_0 is estimated in the interval-based algorithms based on how packed the interval is. But F_0 is estimated based on the probability that a bucket is (non)empty by the bucket-based algorithms. BJKST, LC, and Bloom filter are examples of the bucket-based category. AKMV and MinCount are members of the interval-based class.

The first classification by Flajolet et al. [2008] depends on the observable which an estimation algorithm uses. The second classification by Metwally et al. [2008] is based on the intuition of the algorithm and how it maps the hash values to a bit in the bitmap. We added a third classification based on the core method an algorithm uses to estimate F_0, as explained in the next section.

2.3 Review of twelve cardinality estimation algorithms

After introducing the motivation, the problem, and general approaches to solve it, we provide a new classification and an overview of concrete cardinality estimation algorithms and how they trade the accuracy for runtime and memory consumption. Table 2.2 (page 27) summarizes this section.

2.3.1 Trailing 1s algorithms

This algorithm family uses the number of trailing 1s in the bit pattern observable's bitmap as the core method to estimate F_0.

Flajolet and Martin (FM)

Flajolet and Martin designed the first algorithm to estimate F_0 in a single pass using fewer than one hundred binary words additional storage and only a few operations per elements (what would nowadays be called a data stream scenario) [Flajolet and Martin, 1985]. This algorithm, also known as *probabilistic counting*, uses the observations approach. Flajolet and Martin observed that if a hash function h maps the elements into uniformly distributed integers (binary strings y of length L) over the range $[0 \dots 2^L - 1]$, the pattern $0^k 1 \dots$ appears with probability $1/2^{(k+1)}$.

They formalized this observable as a function $\rho(y)$ that represents the position of the least significant 1-bit in y, i.e., k if $y > 0$ and L otherwise. Then, they recorded these observable values using a bitmap B initialized to all 0. All items with same value set at random the same bit $\rho(y)$ in B to 1. After the algorithm scans the entire dataset, B is independent of any duplication and $B[i]$ is expected to be 1 if there are at least $2^{(i+1)}$ distinct values. If $B[i+1]$ is still 0, F_0 is likely greater than $2^{(i+1)}$ but less than $2^{(i+2)}$.

Therefore, Flajolet and Martin used R, the position of least significant bit that was not flipped to 1 in the bitmap B as an indicator of $log_2(\varphi \cdot F_0)$ with standard deviation close to 1.12. In other words, R is the number of trailing 1s in B. So the estimation $\hat{F_0}$ of the cardinality F_0 is given by $\hat{F_0} = 2^R/\varphi$ where $\varphi = 0.77351$ is a statistical correction factor.

To reduce the variance of R, the FM algorithm is improved to take the average of m runs of the previous procedure using a set of m hash functions to compute m bitmaps, i.e., $\hat{F_0}$ is

$$\hat{F}_0^{FM} = \frac{2^{\overline{R}}}{\varphi} \qquad with \qquad \overline{R} = \frac{1}{m} \cdot \sum_{i=1}^{m} R_i \tag{2.3}$$

Hence, the standard error of the estimator is reduced by a factor of $\mathcal{O}(\sqrt{m})$ to become $\mathcal{O}(1/\sqrt{m})$, yet CPU usage per element processing is multiplied by m. FM's accuracy is directly proportional to the synopsis size namely to the design parameter m. The size L of the bitmap is also an important design parameter and depends on the maximum cardinality N_{max} to which we safely want to count up to, and selected to be larger

than $log_2(N_{max}/m) + 4$. However, error analysis of the FM algorithm is based on the assumption that a specific family of hash functions with some ideal random properties is used.

Probabilistic counting with stochastic averaging (PCSA)

In the same article [Flajolet and Martin, 1985], the authors pointed out that the use of what is called *stochastic averaging* can achieve the same effect as when using direct averaging in the FM algorithm. The result was a new variant of the FM algorithm called *probabilistic counting with stochastic averaging* (PCSA). The algorithm uses the same observable of FM but reduces the processing time per element to $\mathcal{O}(1)$ as well as reducing the synopsis size.

The intuition behind PCSA is to distribute the dataset items into buckets hoping that F_0/m items fall into each bucket. The value R of each bucket, as described in FM, should be close to F_0/m and the average of those values can be used in the right hand side of the Equation (2.3) as $\overline{R}$ to derive a reasonable approximation of F_0/m.

This intuition is implemented using m bitmaps, one per bucket, and a single hash function h that is used to distribute the dataset elements into one of the bitmaps. When PCSA observes a new item x, the $log_2(m)$ least significant bits of the binary representation of $h(x)$ are used to determine the bitmap to be updated and the remaining bits are used to find the observable ρ (same in FM), and then set the corresponding bit to 1 within the previously determined bitmap. So the estimation $\hat{F_0}$ of the cardinality F_0 is given by:

$$\hat{F}_0^{PCSA} = m \cdot \frac{2^{\overline{R}}}{\varphi} \tag{2.4}$$

where φ, L, the size of each bitmap, and $\overline{R}$ are identical to those in FM algorithm. But since each bitmap has seen only $1/m$ of the total distinct values, we multiply by m. PCSA has several advantages over FM: it reduces the cost of FM by using a single hash function and increases the estimation accuracy to an expected standard error of $0.78/\sqrt{m}$.

2.3.2 Leading 0s algorithms

This algorithm family uses the number of leading 0s in the bit pattern observable's bitmap as the core method to estimate F_0. But, the algorithms of this family do not maintain an actual bitmap. Instead they keep only the maximum observable value which equals to the number of leading 0s in the observable's bitmap.

Alon, Martias and Szegedy (AMS)

Alon et al. [1996] provided the first theoretic definition and discussion of the frequent-moments statistics for approximate counting. In their work, they revise the FM algorithm as a randomized algorithm for estimating F_0 and adapt it into the AMS algorithm.

First, they argue that FM was designed assuming an explicit family of ideal random hash functions that could be unrealistic. In consequence, they proposed to use linear hash functions instead. Second, AMS keeps using the same observable $\rho(y)$. But it uses R, the position of most significant bit flipped to 1 in the bitmap B, as an indicator of $log_2(F_0)$. In other words, R is the number of leading 0s in B.

The number of distinct values is likely to be 2^r, if $\rho(y) = r$. Thus, after scanning the entire dataset, one of the observable values hits $\rho(y) \geq log_2(F_0)$. The maximum value of $\rho(y)$, namely R, is a good estimation of $log_2(F_0)$. AMS estimates F_0 by:

$$\hat{F}_0^{AMS} = 2^R \tag{2.5}$$

Finally, AMS does not use a bitmap to record the observable values. Instead, it keeps track of only the maximum observable value R.

The authors proved that AMS guarantees a ratio error of at most c with a probability of at least $1-2/c$ for any $c > 2$. Using m hash functions can further improve the accuracy with the trade-off of increasing the space and time linearly.

FM is still more accurate than AMS [Gibbons, 2007]. The least significant 0-bit in B (R in FM and PCSA) is more accurate than the most-significant 1-bit (R in AMS) to estimate F_0. The reason is that the bit that represents R in AMS can be set by a single outlier hash value. As a result, AMS overestimates F_0 as 2^R, especially when the bits preceding the most significant bit are zeros. AMS and FM share the drawback of performing m hashes for every element.

Bar-yossef, Jayram, Kumar, Sivakumar and Trevisan (BJKST)

Three theoretical algorithms for approximating F_0 are presented in [Bar-Yossef et al., 2002a]. We focus on the third algorithm, because it is the most used and, in a sense, the best one. It is known as the BJKST algorithm, an acronym of the authors' last names. This algorithm is an improvement of the work in [Bar-Yossef et al., 2002b] based on AMS, unified with the Coordinated Sampling algorithm presented by [Gibbons and Tirthapura, 2001].

In essence, BJKST is based on AMS. It uses the same function $\rho(y)$, but does not simply keep track of the maximum value of $\rho(y)$. Instead, it resembles the Coordinated Sampling algorithm and uses a pairwise independent universal hash function h and a buffer B to estimate F_0.

The hash function h guarantees that the probability of $\rho(h(x)) \geq r$ is precisely $1/2^r$ for any $r \geq 0$ as stated in [Alon et al., 1996]. Thus, items $\{x_0, x_2, \ldots, x_n\}$ of the dataset can be assigned to a level according to their $\rho(h(x_i))$ values as following: half of the items have a level equal to 1, a quarter of them have a level equal to 2, and $1/2^r$ have a level equal to r.

The buffer B initially stores all the elements scanned so far and their level is at least $Z = 0$. Whenever the buffer size is larger than a predefined threshold θ, the level Z is

increased by one and all the elements in B with a level of less than Z are removed, and so on.

Unlike the Coordinated Sampling algorithm, BJKST stores pairs $(g(x_i), \rho(h(x_i)))$ instead of keeping the actual value of the element x_i in order to further improve the efficiency of the buffer. g is another uniform pairwise independent hash function. These pairs are stored in array of binary search trees where the j-th entry contains all the pairs in level j.

After one pass over the dataset, BJKST finds the minimum level Z for which the buffer size does not exceed a specific threshold θ. Also, it expects $F_0/2^r$ elements to be in the level Z, i.e., $|B| = F_0/2^Z \leq \theta$. Therefore, $\hat{F}_0$ is

$$\hat{F}_0^{BJKST} = |B| \cdot 2^Z \tag{2.6}$$

BJKST can provide an (ε, δ) approximation scheme of F_0, when the output is the median of running $\mathcal{O}(log(1/\delta))$ parallel copies of the algorithm. Then, $\theta = 576/\varepsilon^2$ for any $\varepsilon > 0$ and $0 < \delta \leq \frac{1}{3}$.

Both BJKST and Coordinated Sampling have the advantages of keeping samples of the data that can be used later. But BJKST improves the efficiency of the buffer, both in space and processing time, as explained above.

LogLog

Durand and Flajolet introduced another AMS-based estimator for F_0, which uses only $\log_2 \log_2(N_{max})$ of memory to estimate cardinalities in range of millions with a relatively high accuracy [Durand and Flajolet, 2003]. This LogLog algorithm uses PCSA's intuition to overcome the overestimation problem in AMS. It improves space usage over PCSA by trading off the accuracy.

The algorithm uses m buckets $B_1, \ldots, B_m$ to distribute the dataset items over them. Then, LogLog uses the AMS approach and maintains R_i for each bucket B_i. Each bucket is responsible for about F_0/m of the distinct elements. Thus, the arithmetic mean $\overline{R}$ of $R_1, \ldots, R_m$ is a good approximation of $log_2(F_0/m)$. The LogLog estimator returns $\hat{F}_0$ with a standard error $\approx 1.3/\sqrt{m}$, as the following:

$$\hat{F}_0^{LogLog} = \alpha_m \cdot m \cdot 2^{\overline{R}} \qquad with \qquad \overline{R} = \frac{1}{m} \cdot \sum_{i=1}^{m} R_i \tag{2.7}$$

Durand and Flajolet [2003] used the correction factor $\alpha_m = 0.39701$ as soon as $m \geq 64$ in their practical implementation.

Whenever a new element x_j is scanned, the algorithm uses the first $k = log_2(m)$ bits of the binary representation of $h(x_j)$ to map the element x_j to a bucket B_i. Then, it updates R_i after comparing its value with $\rho(h(x_j))$, after ignoring the first k bits. Like AMS, LogLog maintains only the value of the maximum R_i, and not a bit vector.

SuperLogLog

SuperLogLog is an optimization of the LogLog algorithm [Durand and Flajolet, 2003]; Durand and Flajolet suggest two improvements. The first one decreases the variance of the $\hat{F_0}$ around the mean, while the second improves the space cost by bounding the size of each R_i. To implement the improvements, SuperLogLog uses two rules: the truncation rule and the restriction rule.

The *truncation rule* refers to discarding the largest 30% of the estimates when averaging R_i to produce the final estimate. In other words, SuperLogLog retains only the $m_0 = \lfloor 0.7 \cdot m \rfloor$ smallest values to compute the truncated sum $\sum^* R_i$. Thus, SuperLogLog estimates F_0 by:

$$\hat{F}_0^{SuperLogLog} = \tilde{\alpha}_m \cdot m_0 \cdot 2^{\overline{R}} \quad with \quad \overline{R} = \frac{1}{m_0} \cdot \sum{}^{*} R_i \tag{2.8}$$

The modified statistical correction factor $\tilde{\alpha}_m = 1.09295$ minimizes the bias [Metwally et al., 2008]. Empirically, this truncation increases the accuracy and bounds the standard error of order $1.05/\sqrt{m}$.

The *restriction rule* is based on an empirically justified fact that R_i values can be restricted to a maximum value of $\lceil log_2(\frac{N_{max}}{m}) + 3 \rceil$, thereby enhance the algorithm synopsis size without any noticeable effect on the estimation accuracy [Durand and Flajolet, 2003].

HyperLogLog

Flajolet et al. [2008] introduced HyperLogLog as a near-optimal successor to LogLog. HyperLogLog uses the same observable, $\rho(y)$, as LogLog and also maintains the maximums R_i. But, it reduces the estimation's variance using harmonic means to estimate F_0 from the maximums R_i.

Based on the same intuition behind LogLog, the harmonic mean $\overline{R}$ of $2^{R_1}, \ldots, 2^{R_m}$ is close to F_0/m. Therefore, HyperLogLog returns an estimation of F_0 as a normalized bias corrected harmonic mean:

$$\hat{F}_0^{HyperLogLog} = \alpha_m \cdot m \cdot \overline{R} \tag{2.9}$$

with

$$\overline{R} = \frac{m}{\frac{1}{2^{R_1}} + \ldots + \frac{1}{2^{R_m}}}$$

and α_m is a bias correction factor where $\alpha_{16} = 0.673, \alpha_{32} = 0.697, \alpha_{64} = 0.709$, and $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$.

The algorithm archives a standard error in the order of $1.04/\sqrt{m}$. The authors' practical results analysis shows that the estimation of F_0 maintains the theoretical standard error in the range $]\frac{5}{2} \cdot m, \frac{2^{32}}{30}]$ for any $m \in \{2^4, \ldots, 2^{16}\}$.

Two corrections are introduced to deal with the $\hat{F}_0^{HyperLogLog}$ values that fall outside the specified range, either in the *small range* (i.e., $\leq \frac{5}{2} \cdot m$) or in the *large range* (i.e., $> \frac{2^{32}}{30}$).

The problem in small range is the presence of nonlinear distortions. The source of the bias is the high number V of $R_i = 0$ in the harmonic mean when n is small compared to m. The small range correction uses LC to estimate F_0 from the maximums R_i when $V > 0$ as:

$$\hat{F}_0^{HyperLogLog*} = m \cdot log(\frac{m}{V}) \quad (2.10)$$

In the large range the cardinality is reaching 2^{32}, which causes an increase in the hash collisions due to 32-bit hash function used by HyperLogLog. So, the algorithm applies a correction to the estimation and returns:

$$\hat{F}_0^{HyperLogLog*} = -2^{32} \cdot log(1 - \frac{\hat{F}_0^{HyperLogLog}}{2^{32}}) \quad (2.11)$$

Hence, HyperLogLog is a bit-pattern observable algorithm. Yet, it can also be viewed as order statistics observable algorithm, because $1/2^{R_i}$ is an estimation of $min(B_i)$ up to a factor at most 2. The authors argue that HyperLogLog is near optimal, because its estimation standard error is near $1/\sqrt{m}$, the lower bound for accuracy achievable by order statistics algorithms.

HyperLogLog++

HyperLogLog++ is a revision of the HyperLogLog algorithm [Heule et al., 2013]. The authors suggest a series of changes to improve the original algorithm's estimation accuracy and reduce the space cost. The development of this algorithm was driven by the need to accurately estimate cardinalities well beyond 10^9, as well as small cardinalities and to efficiently adapt memory usage to the cardinality. The authors present three improvements, which can be applied together or independently to fit the need of the application.

First, HyperLogLog++ uses a 64-bit hash function as a replacement to the high range correction in the original algorithm with low additional cost in memory. This increases the size of each R_i by only one bit, but it enables to estimate cardinalities approaching 2^{64} before the hash collisions start to increase.

Second, the authors experimentally found a bias correction method that works effectively up to $n = 5 \cdot m$. They estimate the bias of F_0 from $\hat{F_0}$ using k-nearest neighbours interpolation with the empirically determined values. They further combine this bias correction of the estimation with LC. LC is used to correct estimations that are lower than θ, an empirically determined threshold.

Third, HyperLogLog++ develops a *sparse representation* to avoid the cases where $n \ll m$ and most of the R_i's are never used. This representation is identical to the one used in BJKST, where the algorithm stores pairs $(idx, \rho(y))$. But, HyperLogLog++

switches back to the original *dense representation* whenever maintaining this list consumes more memory than the original memory consumption. As a result, the memory consumption is reduced for small cardinalities with small runtime overhead to maintain the new representation.

2.3.3 K-th minimum value algorithms

This algorithm family uses order statistics as their observable, specifically the k-th minimum value.

MinCount

The MinCount algorithm was introduced by Giroire [2009] as a generalization of the first algorithm presented by Bar-Yossef et al. [2002a], which is where also BJKST was introduced. Like the original algorithm, MinCount is an interval-based algorithm that uses the *k-th minimum value order statistics observable* (KMV) to estimate the density of the interval, which is in turn used to estimate F_0. In other words, MinCount considers the hashed values as a set of independent uniformly distributed real numbers in the interval $[0, 1]$ with repetitions, i.e., an *ideal multiset* $\acute{E}$.

The algorithm's main idea is that the first minimum of $\acute{E}$ is an indication of $1/(F_0+1)$. However, the inverse of this minimum has an infinite expectation. MinCount avoids this by using two new aspects: It combines $M^{(k)}(k \geq 2)$ the k-th minimum of $\acute{E}$ and a sublinear function of $1/M^{(k)}$, such as its logarithm or square root, as a replacement of the first minimum and its direct inverse alone.

Furthermore, MinCount reduces the standard error using the stochastic averaging like in PCSA. So, the hashed values interval is divided into m buckets. A hash value $h(x)$ is mapped to the bucket i if $(i-1)/m \leq h(x) < i/m$. For each bucket, the values $M_i^{(k)}$ are maintained for $i = 1, \ldots, m$. MinCount's best estimate of F_0 is using $k = 3$ and the logarithm function as follows:

$$\hat{F}_0^{MinCount} = m \cdot \left(\frac{\Gamma(k - \frac{1}{m})}{\Gamma(k)} \right)^{-m} \cdot e^{\overline{R}} \tag{2.12}$$

$$with \qquad \overline{R} = -\frac{1}{m} \cdot \sum_{i=1}^{m} ln(M_i^{(k)})$$

where Γ is the Euler Gamma function. The standard error of this estimation is up to $1/\sqrt{M}$ using $M = k \cdot m$ units of storage.

AKMV

Beyer et al. also revised the first algorithm proposed by Bar-Yossef et al. [2002a] to introduce their unbiased version of F_0 estimator based on KMV order statistics [Beyer et al., 2007, 2009]. The authors provided several estimators of F_0 in two scenarios:

(1) when the dataset consists of only one partition, which is what we study in this survey, and (2) when the dataset is split into partitions and the estimation is obtained in parallel with the presence of multiset operations.

The original algorithm uses the k-th smallest hash value to estimate K/F_0. Beyer et al. [2009] showed that the original estimator overestimates F_0 and is biased upwards towards F_0. To lower this bias, the AKMV estimator is given by:

$$\hat{F}_0^{AKMV} = \frac{(k-1)}{M^{(k)}} \tag{2.13}$$

where $F_0 > k$, otherwise the algorithm finds the exact F_0. If F_0 is expected to be large, the suitable synopsis size k can be determined based on the error bounds. AKMV's relative error is bounded to $\sqrt{2/(\pi \cdot (k-2))}$.

For the second scenario, i.e., to support the multiset operations among the synopses of the partitions, the authors introduce the AKMV synopsis and a corresponding F_0 estimator, to estimate F_0 of each partition as well as of the whole dataset. In addition to the k minimum hash values, AKMV maintains k counters. Each counter contains the multiplicity of the corresponding element in the k minimum hash values set. The $\hat{F}_0^{AKMV}$ estimator was generalized to estimate F_0 for compound partition created from disjoint partitions by multiset operations.

2.3.4 Linear synopses based estimators

The core method for estimating F_0 for this algorithm family is how packed or empty its linear synopsis is. This family uses a uniform hash function(s) to distribute the items over the linear synopsis or to group them into buckets. Then, the algorithm estimates the cardinality based on the density of the synopsis.

Linear Counting (LC)

Whang et al. [1990] present a probabilistic algorithm for estimating the number of distinct values in a dataset called Linear Counting (LC). This algorithm is a straightforward application of the bitmap of hash values approach. LC maintains a bitmap B of size b, in which all entries are initialised to 0.

LC is neither a logarithmic hashing algorithm nor a logarithmic counting algorithm. In contrast, it is a linear counting algorithm that applies a uniform hash function h to each item from the dataset x. Then, $h(x)$ maps the item *uniformly* to a bucket in the bitmap and sets it to 1, i.e., $B[h(x)] = 1$.

After hashing the entire dataset, if there were no collisions the number of 1-bits in B would be F_0. But F_0 can be estimated based on the probability that a bucket is empty. Let V_n denote the fraction of empty buckets in the bitmap. It is a good estimation of this probability. The expected probability of a bucket being empty is given by $e^{-n/b}$. As a result, F_0 is estimated using maximum likelihood estimator:

$$\hat{F}_0^{LC} = -b \cdot ln(V_n) \tag{2.14}$$

The size b of the bitmap is defined in terms of a constant called load factor t as $b = F_0/t$. Whang et al.'s analysis reveals that using $t \leq 12$ provides $\hat{F}_0$ with 1% standard error. This fact reduces the synopsis by a factor of t. That leads to the main limitation of LC: it needs some prior knowledge of F_0 to determine the size of B. Practically, the upper bound of cardinality n_{max} is used when creating B instead of F_0. Thus, a linear space of order $\mathcal{O}(n_{max})$ is the main drawback of LC, when we have limited memory or datasets of high cardinalities. Nevertheless, LC is a simple algorithm that can provide a highly accurate estimation $\hat{F}_0$, if one chooses the right load factor. The standard error of $\hat{F}_0^{LC}$ is $\sqrt{(e^t - t - 1)/(t \cdot n_{max})}$.

Bloom filter (BF)

The main source of the algorithm LC's estimation error are hash collisions in the bitmap. Bloom filters can reduce collisions using m independent hash functions. Unlike LC, each element is mapped to a fixed number of bits $\leq m$, i.e., $B[h_i(x)] = 1$.

The standard Bloom filter is designed to maintain the membership information rather than a statistical information about the underlying dataset. But one can count the distinct elements in a multiset by combining a Bloom filter with a counter, which is incremented whenever an element is not in the filter. The value of the counter can never be larger than the exact cardinality due to the Bloom filter's nature, but hash collisions can cause it to underestimate F_0.

Bloom filters have been used effectively for cardinality estimation. Like LC, Swamidass and Baldi introduced an estimator of the population of Bloom filter using X as the number of bits set to 1 in the filter [Swamidass and Baldi, 2007]. Intuitively, after inserting all the elements of the dataset into a Bloom filter, the number of elements in a Bloom filter is in fact F_0 of the represented dataset. Given a Bloom filter of size b with m hash functions, F_0 can be estimated by:

$$\hat{F}_0^{BF1} = -\frac{b}{m} \cdot ln(1 - \frac{X}{b}) \tag{2.15}$$

Papapetrou et al. [2010] proposed a probabilistic approach to estimate F_0 of a dataset from its standard Bloom filter representation. This approach estimates the number of elements in a Bloom filter based on its density and requires only X and the configuration of the Bloom filter (b and m).

They estimate F_0 by the maximum likelihood value for the number of hashed elements:

$$\hat{F}_0^{BF2} = \frac{ln(1 - \frac{X}{b})}{m \cdot ln(1 - \frac{1}{b})} \tag{2.16}$$

Table 2.2: Error-guarantees of the twelve algorithms

Algorithm	Error	Notes
FM	Std. err. = $(1/\sqrt{m})$	m: Number of hash functions
PCSA	Std. err. = $0.78/\sqrt{m}$	m: Number of bitmaps
AMS	Ratio err. $< c$ with probability $> 1 - 2/c$	$c > 2$
BJKST	(ε, δ) Approximation Scheme	Relative err. $\varepsilon > 0$ $0 < \delta \leq 1/3$
LogLog	Std. err. = $1.3/\sqrt{m}$	m: Number of maximums R_i
SuperLogLog	Std. err. = $1.05/\sqrt{m}$	m: Number of maximums R_i
HyperLogLog	Std. err. = $1.04/\sqrt{m}$	m: Number of maximums R_i
HyperLogLog++	Smaller by factor 4 compared to HyperLogLog for cardinalities up to 12,000 [Heule et al., 2013].	Precision = 14 Sparse representation precision = 25
MinCount	Std. err. = $1/\sqrt{k \cdot m}$	k: order of the used minimum m: Number of buckets
AKMV	Relative err. $\approx \sqrt{2/(\pi \cdot (k-2))}$	k: order of the used minimum
LC	Std. err. = $\sqrt{\frac{(e^t - t - 1)}{(t \cdot n_{max})}} = 0.01$ for $t \geq 12$	t: load factor n_{max}: Upper bound on n
BF	Relative err. ≤ 0.04 [Papapetrou et al., 2010]	BF density = 0.9

Their evaluation results show that the Bloom filter configuration affects the estimation accuracy – larger Bloom filter provides higher estimation accuracy. Bloom filters with fewer hash functions exhibit a more accurate cardinality estimation. Bloom filter shares with LC the same limitation of the need of a prior knowledge of the maximum cardinality in order to choose the suitable size of the filter.

Count-Min, not to be confused with MinCount, is a Bloom filter-like sub-linear synopsis, which estimates the dataset item's frequencies [Cormode and Muthukrishnan, 2005]. Count-Min is not originally designed to track the number of distinct values, but to solve problems such as determining quantiles and heavy hitters. However, Cormode [2009] pointed out that Count-Min sketches could be updated using FM-like synopses to achieve this goal.

To summarize, we presented an overview of the state-of-the-art cardinality estimation algorithms. In the following section, we compare the described algorithms and benchmark their accuracy, runtime, and memory consumption. Table 2.2 summarizes the error guarantees of the algorithms presented in this section.

2.4 Comparative experiments

In the papers where they had been introduced, most of the twelve algorithms are accompanied by a theoretical analysis of how well they estimate F_0 in terms of error and space

bounds. Nevertheless, these analyses suffer from some shortcomings. The $\mathcal{O}()$ notation in space bounds hides the actual space used for maintaining hash functions and data structures. Furthermore, there is no unified error metric or hashing assumption among the algorithms. To decide on a suitable algorithm for a given use case one needs more information.

In this section we experimentally compare all twelve F_0 estimation algorithms to analyze and better understand their behavior using a unified error metric, the same amount of memory, and the same hash function. First, we describe the experimental setup and the implementation details. Then, we briefly evaluate a sampling-based algorithm. We compare the algorithms' accuracies among each other and per algorithm family. Next, we study the correlation between runtime, exact F_0, and dataset size. Finally, we measure memory consumption of these algorithms and report minimum memory needed to run each algorithm.

2.4.1 Experimental setup

Hardware

We performed all experiments on a Dell PowerEdge R620 server running CentOS 6.4. It has two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) processors, 128 GB DDR3-1600 RAM and a 4 TB RAID-5 storage. We implemented all algorithms as single-threaded Java applications using OpenJDK 64-Bit Server VM 1.8.0_111-b15.

Implementations

To guarantee a unified test environment when comparing the twelve F_0 estimation algorithms, we implemented them for the Metanome data profiling framework [Papenbrock et al., 2015a]. Metanome is a standard framework decoupled from the algorithms[2]. It provides basic functionalities, such as input parsing and performance measurement. In addition to various of discovery algorithms for complex metadata, such as keys or functional dependencies, we extended Metanome to support basic statistics algorithms, including F_0 estimation.

To further unify our comparison we need to avoid the significant impact of the used hash function on the runtime and the estimation accuracy. Thus, we implemented all algorithms using the same hash function, namely MurmurHash[3]. We chose MurmurHash based on the results in [Singh and Tirthapura, 2015], where the authors showed that PCSA, LC, and LogLog yield the fastest and most accurate F_0 estimation when using MurmurHash compared to Jenkins, Modulo congruential hash, SHA-1, and FNV.

The next implementation decision was whether to use a 64-bit or 32-bit version of MurmurHash. Nowadays, in the era of "Big Data", it is important to estimate cardinalities of over 10^8. The algorithms based on an observable of the hash values are limited by the number of bits used to represent these hash values. For linear synopses, using 64-bit hash functions reduces collisions in the case of large datasets. As a result, we implement

[2] `www.metanome.de`

[3] `https://sites.google.com/site/murmurhash/`

all algorithms using the 64-bit MurmurHash version. We made an exception of 32-bits for AKMV and MinCount, because they both use the k-minimums of the hashed values and using 64 bits adds an overhead without improving the algorithms' counting ability.

We counted the exact value of F_0 using the "JavaHashSet". Unless stated otherwise, all algorithms were configured to produce theoretical (standard/relative) errors of 1% according to Table 2.2. LC and Bloom filter use the number of tuples in the dataset as n_{max}. Bloom filter is implemented as a standard Bloom filter with four bits per element and three hash functions to minimize the false positive rate and preserve the membership test ability of the filter. For a detailed experimental evaluation of the influence of Bloom filter length, number of hash functions, and number of blocks, on estimation accuracy, refer to [Papapetrou et al., 2010].

Our re-implementations, all datasets, and results are available on our repeatability page[4].

Datasets

To benchmark the estimation accuracy and runtime of the considered F_0 estimation algorithms, we have run them over real-world datasets as well as synthetic datasets.

The 90 synthetic datasets were generated by the Mersenne Twister random number generator [Matsumoto and Nishimura, 1998]. For each specific cardinality, we generated ten independent datasets using different seeds for each of them and report their average runtime and estimation error. The exact cardinalities were made to be the powers of 10, starting with 10 up to 10^9.

By *dataset cardinality*, we always refer to the number of distinct values in the dataset, while the *dataset size* is the overall number of elements. Table 2.3 shows our real-world datasets, chosen based on tuple count, i.e., how large the dataset size is, and the variety of columns cardinalities as illustrated in Figure 2.2.

Table 2.3: Real-world dataset characteristics

Dataset	[#] **Attributes**	[#] **Tuples**
NCVoter	25 (of 71)	7,560,886
Openadresses-Europe	11	93,849,474

NCVoter is a collection of North Carolina's voter registration data[5]. We used the first 25 columns to perform experiments. The Openadresses dataset is a public database connecting the geographical coordinates with their postal addresses[6].

To avoid the possibly misleading results caused by NULL semantics, all NULL values are discarded by cardinality estimation algorithms and while determining datasets size and exact cardinality (Figure 2.2).

[4] `https://hpi.de/naumann/projects/repeatability/data-profiling.html`
[5] `https://www.ncsbe.gov/data-statistics`
[6] `https://openaddresses.io/`

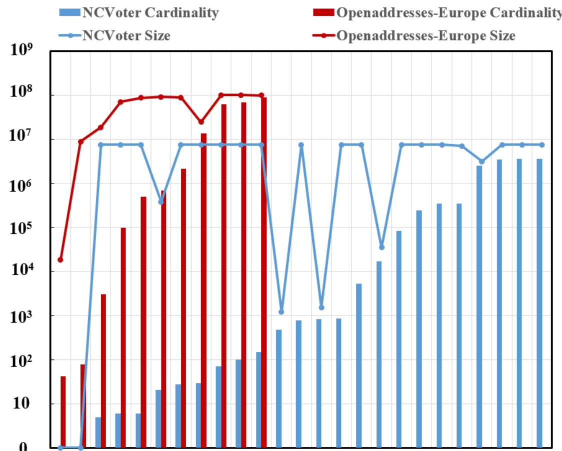


Figure 2.2: Exact cardinality range of the real-world datasets columns and corresponding column size.

Evaluation metrics

We allocate the same memory capacity to all algorithms and evaluate their performance regarding runtime and estimation accuracy. We use relative error as the measure of estimation accuracy as described in Section 2.1. The total time taken by an algorithm to process all the data elements and estimate F_0 is considered as its runtime. Runtime and relative error of the real-world datasets are averaged over ten runs using the same dataset. Runtimes and relative errors are averaged among ten synthetic datasets for each specific cardinality.

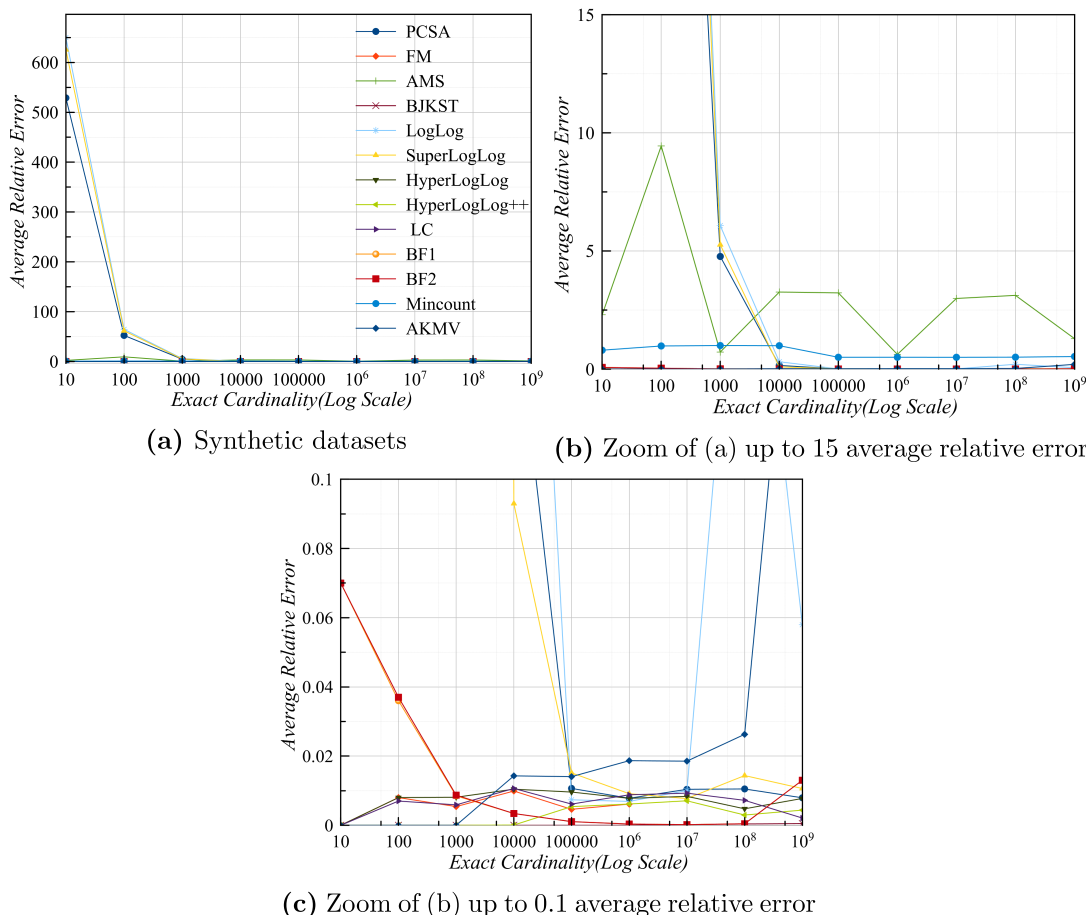
2.4.2 Accuracy experiments

This section compares the accuracy of the twelve algorithms and how they scale with the exact cardinalities of the datasets. We limited the Java Virtual Machine (JVM) to 100 GB and ran each algorithm on each dataset with runtime limit of two hours. Figures 2.3 and 2.4 illustrate the change of the algorithms' relative error for each exact F_0 of the input dataset for synthetic and real datasets. The runtimes of this experiment set are depicted in Figures 2.6 and 2.7, and are discussed in the next section.

Accuracy comparison among all algorithms

For datasets with a low number of distinct values (up to 1,000), all the logarithmic hashing algorithms that use stochastic averaging as a method for accuracy boosting, namely **PCSA**, **LogLog**, and **SuperLogLog**, extremely overestimate F_0.

This effect is a consequence of stochastic average magnification of estimation by m. From Table 2.2 (page 27), we can tell that as m increases, the upper bound of the standard error of the over-all algorithm decreases. The accuracy of PCSA, LogLog, and SuperLogLog increases up to this bound for larger cardinalities. Still, the bias of PCSA



(a) Synthetic datasets

(b) Zoom of (a) up to 15 average relative error

(c) Zoom of (b) up to 0.1 average relative error

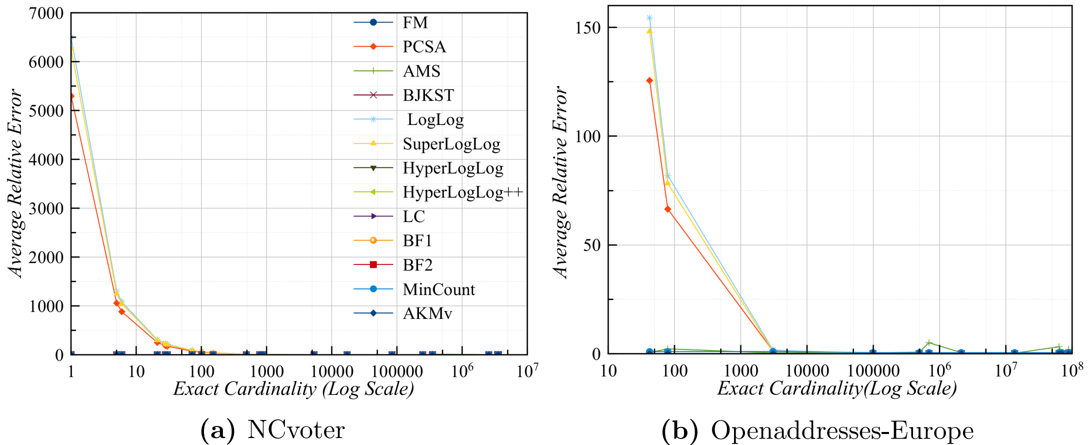
Figure 2.3: Accuracy of the twelve F_0 estimation algorithms over 90 synthetic datasets

is slightly less than that of the others due to the use of trailing 1s of the hash value binary strings as observable. Figure 2.3a clearly shows this observation.

Overall, we see that all algorithms perform similarly well for larger cardinalities. However, taking a closer look, Figure 2.3b makes it clear that **AMS** and **MinCount** perform worse than the rest, even for large cardinalities. We note that AMS has a high variance and is presented as a theoretical algorithm. Our measurements show this variance in practice and show how the LogLog algorithm solved this problem, at least for large cardinalities, using stochastic averaging.

As is clear from Figure 2.3c, **BJKST** outperformed all the other algorithms. The error guarantee of BJKST was even better than the theoretical lower bounds (i.e., relative error was always far less than 1%).

The second best algorithm after BJKST was **Bloom filter** with error measures close to or equal to zero for most of the cardinality range. Noticeably, the relative error of Bloom filter is inversely proportional to the exact cardinality of the dataset. In other words, when the ratio of Bloom filter size over the dataset cardinality is relatively low, the estimation accuracy is improved until the point when the Bloom filter is full and



(a) NCvoter

(b) Openaddresses-Europe

Figure 2.4: Accuracy of the twelve F_0 estimation algorithms on real-world datasets

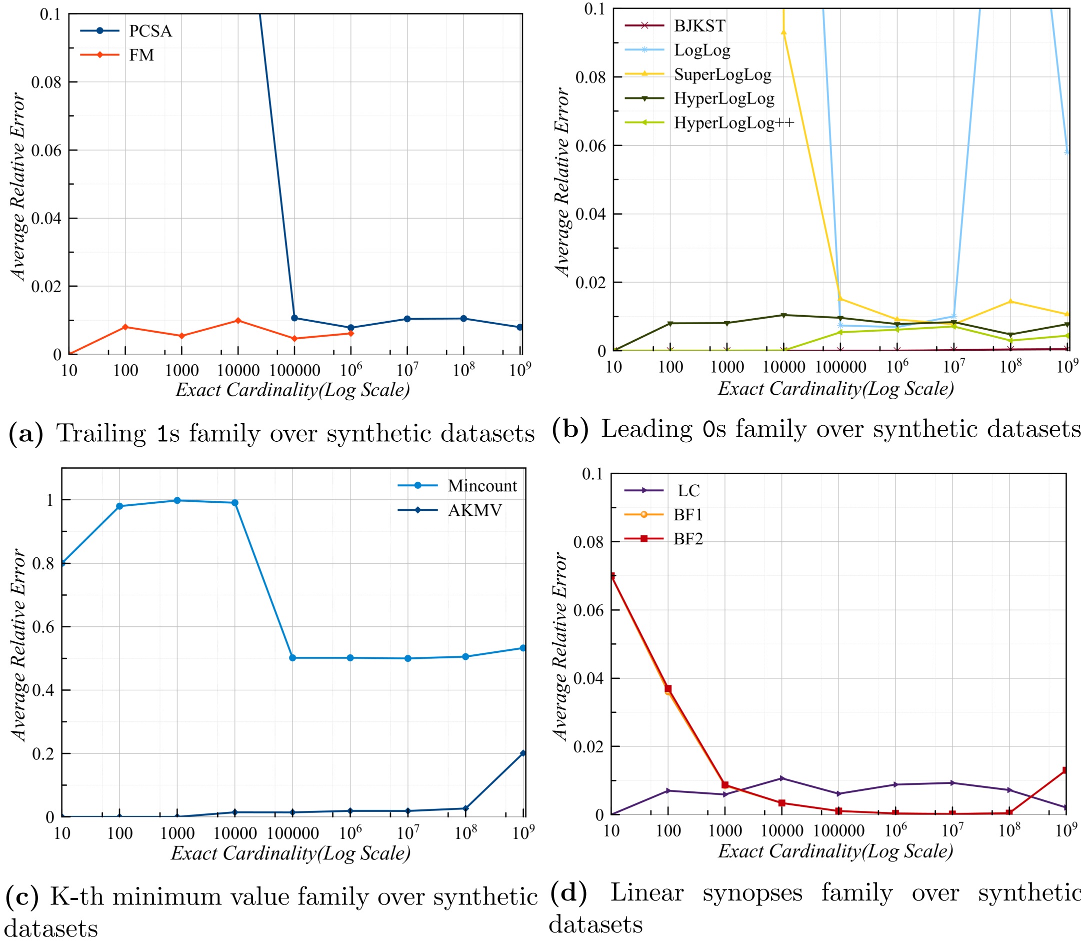
the error rises again. Obviously, this accuracy comes at a cost: we analyze runtime in Section 2.4.3 and memory consumption in Section 2.4.4.

HyperLogLog++ maintained a good estimation accuracy with relative error below 0.008, regardless how many distinct values the dataset has. The bias correction implemented in HyperLogLog++ caused a tangible enhancement in estimating the cardinalities of datasets with small F_0. **FM** is a strong competitor of HyperLogLog++, but it exceeded the two hour runtime limit for datasets with $F_0 > 10^6$.

According to Metwally et al. [2008] experimental results, **LC** was the most accurate F_0 algorithm, beating LogLog, SuperLogLog, FM, PCSA, MinCount, and BJKST. The authors studied the accuracy change with only much smaller datasets and with differing space usage, which is a different setting than in our experiments, which uses different cardinalities. In fact, in our setting, LC also beat LogLog, SuperLogLog, PCSA, and MinCount, but not FM and BJKST.

HyperLogLog provided a very good, and in particular stable estimation. Its use of LC to estimate small cardinalities explains the similarity of the behavior of HyperLogLog and LC for $F_0 < 10^6$. **AKMV**'s accuracy went down steadily with the increase of the dataset cardinality. Beyer et al. [2007] experimentally showed that AKMV is significantly more accurate than SuperLogLog. Their measurements went up to cardinalities of 10^7. In our experiments we observe that AKMV loses this advantage for cardinalities over 10^5 (but it remains to be more efficient than SuperLogLog). In contrast, PCSA, LogLog, and SuperLogLog performed better for high cardinalities than for lower ones.

We also tested the accuracy of the twelve F_0 estimation algorithms on real-world datasets, as shown in Figures 2.4a and 2.4b. A very poor performance can be observed for PCSA, LogLog, and SuperLogLog (the only algorithms appear clearly in the Figures) for columns with low cardinalities on both datasets. More than half of the columns of the NCVoter dataset, as well as only two of the Openaddresses-Europe dataset have cardinalities below 1,000 (Figure 2.2). So, overall we can draw the same conclusion as with our experiments on synthetic datasets.



(a) Trailing 1s family over synthetic datasets **(b)** Leading 0s family over synthetic datasets

(c) K-th minimum value family over synthetic datasets **(d)** Linear synopses family over synthetic datasets

Figure 2.5: Average relative error of the algorithm families over 90 synthetic datasets

For the remaining columns of NCVoter, MinCount had the highest average error around 0.5. All the other algorithms provided a very good accuracy with average relative error less than 0.02. The same observation is also valid on Openaddresses-Europe.

Accuracy comparison per algorithm family

Because accuracy covered a very wide range of values, it was not easy to compare all the algorithms together. To extend our discussion to another perspective, this section validates what we know about each family, discusses the advantages and disadvantages of its algorithms, and identifies the most accurate candidate for each algorithm family. Figure 2.5 shows our findings.

As shown in Fig. 2.5a, in the **trailing 1s** family, FM is more resilient to dataset cardinality than PCSA, but it is impractical with respect to runtime as we show in the next section.

A remarkable variance concerning their accuracy change among the **leading 0s** family algorithms is illustrated in Figure 2.5b. The main disadvantage of this algorithm family is that its algorithms are sensitive to hash value outliers (LogLog substantiates this observation).

Each algorithm enhances its accuracy by using a dedicated boosting method to combine results from m instances of the algorithm or/and correcting specific bias ranges experimentally. LogLog and SuperLogLog use the stochastic averaging method. HyperLogLog and HyperLogLog++ use harmonic means. SuperLogLog and HyperLogLog++ add also a specific bias correction tuned by experimental observations, which explains why they outperformed LogLog, and HyperLogLog, respectively. Furthermore, we can conclude that using the harmonic means method has an appreciable effect in reducing the impact of hash-value outliers and in boosting the overall estimation accuracy without adding a time overhead (Section 2.4.3). Interestingly, BJKST is the best member of this family, although it does not use any additional boosting method to overcome the problem of outliers with added cost for maintaining samples of the original dataset.

AKMV from the **K-th minimum value** family is another example of an algorithm using only one instance defeating other algorithms that combine results of multiple instances. MinCount is designed to use the stochastic averaging method to step-up its accuracy. Still, the fairly simple algorithm AKMV is more accurate than MinCount, as shown in Figure 2.5c.

As anticipated in the **Linear synopses** family, Bloom filter beats LC due to their adoption of multiple hash functions resulting in a lower hash collision rate (Figure 2.5d). An expected consequence is that LC is faster than Bloom filter as we show in the next section. Nevertheless, the previous knowledge of maximum F_0 in order to fairly tune these algorithms is the drawback of this family.

2.4.3 Runtime behavior experiments

To compare the runtimes of the twelve algorithms, we recorded the runtime of the accuracy experiments in the previous section. We used a runtime limit of two hours. This period was not enough to count exactly the distinct values of the generated datasets with cardinalities above 10^9, nor for the FM algorithm to finish: despite its high accuracy, FM exceeded the time limit for datasets with F_0 over 10^6, both in synthetic and real-world datasets. Figures 2.6 and 2.7 represent in log scale the runtimes for synthetic and real-world datasets, respectively.

General speaking, for the synthetic datasets the runtimes of all algorithms scale quadratically with synthetic dataset size (equaling their cardinality here), FM being the slowest. All twelve algorithms apply hashing on every single element of the dataset, and hashing constitutes the majority of each algorithm runtime. Accordingly, the runtime mainly depends on the size of the dataset, not its cardinality. As the synthetic datasets' sizes are identical to their F_0, the correlation between dataset size and runtime is obvious.

In contrast, real-world dataset columns consist of duplicated items and have different size within the same dataset due to removal of NULL values. For example, more than half

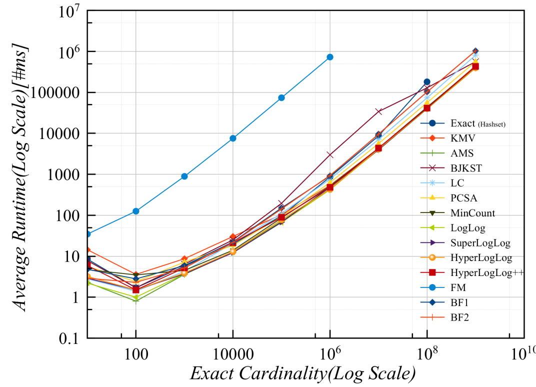
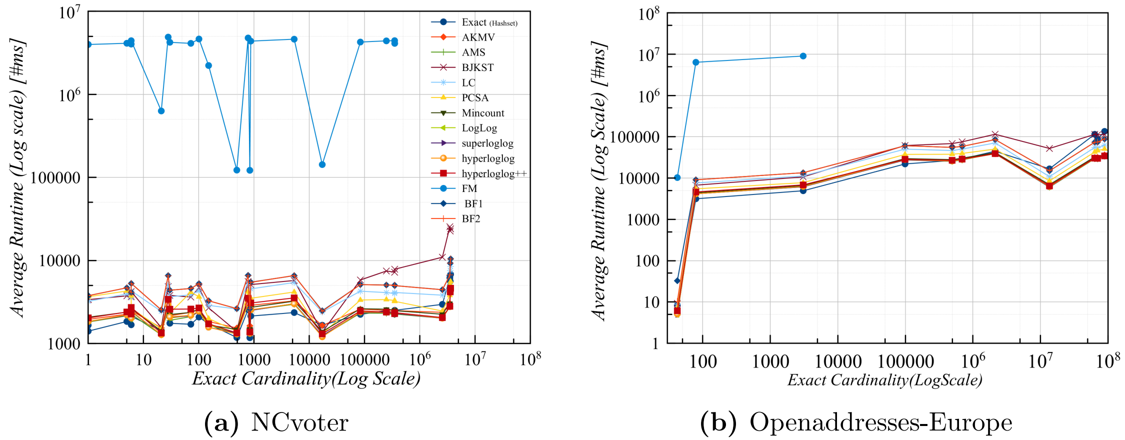


Figure 2.6: Runtime behavior of the twelve cardinality estimation algorithms on synthetic datasets.



(a) NCvoter

(b) Openaddresses-Europe

Figure 2.7: Runtime behavior of the twelve cardinality estimation algorithms on real-world datasets

of the elements in the tenth column of openaddresses-Europe are NULL values. Despite the high F_0 value, this column has fewer elements than columns with a lower cardinality, explaining the dip at around 10^7. Figure 2.7a shows a similar behavior for the NCVoter dataset. Comparing it with NCVoter columns size in Figure 2.2, we notice the influence of column size of runtime of all algorithms.

In addition to dataset size, two factors have a significant impact on runtime: the number of used hash functions and the maintained data structures (synopsis type). The disadvantage of using m hash functions is noticeable in FM's runtime behavior. It is slower by a factor of two than all other algorithms, regardless of input dataset cardinality, but still follows the same influence of dataset size. The second slowest algorithm is BJKST due to the overhead of keeping samples of the dataset and using a second hash function in order to comprise its synopsis. Our measurements revealed that FM and BJKST were slower than counting the exact F_0 using a hash table, when the JVM heap size was limited to 100 GB (i.e., large memory budget).

LC hashes each element from the dataset once, so it is obviously faster than Bloom filter which uses three hash functions. Both LC and Bloom filter need large synopses, proportional to dataset size. Except FM, PCSA is the only logarithmic hashing algorithm that keeps track of all the observable values (i.e., $\rho(y)$), making it slightly slower.

In summary, all the rest of the algorithms ran in roughly the same time without a critical difference. HyperLogLog++, HyperLogLog, AKMV, and LC are the best algorithms in terms of accuracy and runtime using a large memory budget.

2.4.4 Memory consumption experiments

In the previous experiments, we discussed the accuracy and runtime behavior using a large memory budget (JVM was limited to 100 GB). It is a key requirement for cardinality estimation to efficiently use all available memory. In some cases, a dataset is so massive that count-distinct queries could not be run within available memory. For example, out-of-memory was the error of a non-negligible part of count-distinct queries in the PowerDrill system [Heule et al., 2013]. In other cases, such as stream processing, the available memory is typically orders of magnitude smaller than the input [Garofalakis et al., 2016].

Therefore, we evaluated the memory efficiency of the twelve algorithms. We determined the heap size that was used by each algorithm to estimate F_0 of each dataset as precise and fast as with large memory. We kept a runtime limit of two hours and all algorithms configured to guarantee 1% estimation error. We report the minimum heap size of ten runs for each data point.

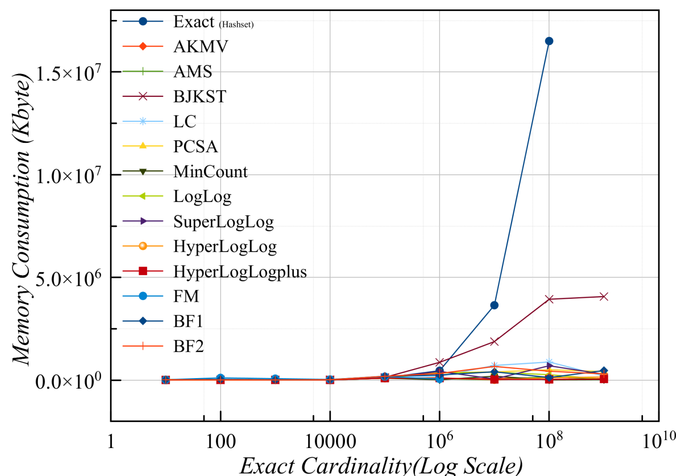


Figure 2.8: Memory consumption of the twelve cardinality estimation algorithms on real-world datasets.

Among all algorithms, only the hash table required a linear (and thus unacceptable) memory consumption. For example, a heap size of 16.5 GB is required to exactly estimate a cardinality up to 10^8. All other algorithms have similar sub-linear behavior. Among those, BJKST has the highest constant factor. As described in Section 2.3, BJKST stores

samples of the data as part of its mechanism which explains the outstanding memory consumption by BJKST compared to other approximation methods, especially to those in the same family. HyperLogLog++ has the lowest constant factor and needs no more than 52 MB heap size.

We ran the same experiments over real-world datasets and also observed an increase in the constant factor in correlation with dataset size.

2.4.5 Sampling-based experiments

Sampling has an inherent difficulty to accurately estimate the number of distinct values (Section 2.2). However, for the sake of completeness, we briefly evaluated *Guaranteed-Error Estimator(GEE)* [Charikar et al., 2000] as an example of a sampling-based cardinality estimator. GEE estimates the number of distinct values based on the frequency of values within the uniformly and randomly sampled items from a column or a dataset. The GEE estimator of the cardinality of a multiset of size n is defined as follows:

$$\hat{F}_0^{GEE} = \sqrt{\frac{n}{r}} \cdot f_1 + \sum_{j=2}^{r} f_j \tag{2.17}$$

Where r is the number of sampled items and f_j is the number of items with frequency j within the sample.

For this experiment, we used *Reservoir sampling* without replacement to randomly sample $20\%, 40\%, 60\%$, and 80% of each column. Table 2.4 shows the experimentally observed effect of different sampling rates on the average GEE estimation error over all columns per dataset. To produce a cardinality estimation with 1% relative error, GEE needs to sample more than 90% of the dataset. We also observed that when GEE is applied to datasets with duplicated values, the estimation error is less.

Table 2.4: GEE average estimation relative error vs. sampling rate

Dataset	**Sampling rate**				
	20%	**40%**	**60%**	**80%**	**100%**
Synthetic	0.54	0.43	0.4	0.2	0
NCVoter	0.26	0.19	0.17	0.07	0.00002
Openadresses	0.28	0.2	0.19	0.09	0.00001

GEE runtime noticeably increases with the size of the dataset, but only slightly with the sampling rate. The main drawback of GEE is its memory consumption. A GEE synopsis consists of both sampled values and their frequencies. For a single column, GEE needs a minimum heap size of at least 13 GB and 35 GB to guarantee an estimation error below 1% on NCVoter and Openadress-Europe, respectively.

2.5 Summary

Single-column data profiling provides a basic understanding of relational data by generating a diverse set of descriptive metadata about each column. Column's cardinality is one of the basic statistics about a column. It refers to the number of distinct values in the respective column. Efficiently estimating the cardinality of a column, a dataset, or a stream is a widely studied problem. We reviewed and discussed twelve of the most important algorithms addressing this task.

We have confirmed that some preliminary solutions, such as sampling and hash tables, are valid only when one can scale up the available computational resources. Both sampling and hash tables have the disadvantages of linear memory consumption and quadratic runtime with dataset size.

Our work has led us to conclude that none of the twelve estimation algorithms is clearly the best for all datasets and all scenarios. For a given accuracy, dataset size is obviously the main factor, affecting all the algorithms' runtime and memory consumption.

We have categorized the algorithms into four families: Count trailing 1s, count leading 0s, k-th minimum value, and linear synopses. We showed that FM, BJKST, AKMV, and Bloom filter are the best among their families, respectively. However, FM needs an extremely high runtime. BJKST and Bloom filter, on the other hand, have a high memory consumption. But AKMV survived for very large cardinalities with low memory consumption and runtime. For datasets with expected small cardinalities, PCSA, LogLog, SuperLogLog are not recommended due to their overestimation problem. Finally, HyperLogLog, AKMV, and LC are efficient over all cardinality ranges by all means.

This study has investigated only single-threaded implementations of the algorithms. However, several algorithms have characteristics that make them ready for parallelization and distributed environments. We can divide them into three categories: (1) Algorithms whose partial results can be easily merged, such as PCSA, AMS, and all their modifications. If the same hash function was used by all threads/nodes, bit-wise OR-operation among their bitmaps can lead to the same final bitmap of a single thread. (2) Algorithms running several copies of the same algorithm or use several hash functions to improve their accuracy, such as FM, MinCount, and Bloom filters. These can be distributed in a straightforward manner. (3) Algorithms allowing set operations like intersections or unions, such as AKMV. In conclusion, there is an ample room for future work to evaluate parallel implementations of these algorithms.

Chapter 3

Discovering Missing Column Headers

The *header* of a column is among the most relevant types of single-column metadata for relational tables, because it provides meaning and context in which the data is to be interpreted. Headers play an important role in many data integration, exploration, and cleaning scenarios, such as schema matching, knowledge base augmentation, and similarity search. Unfortunately, in many cases column headers are missing, because they were never defined properly, are meaningless, or have been lost during data extraction, transmission, or storage. For example, around one third of the tables on the Web have missing headers; it is even worse for open data tables, which are seldomly supplied with meaningful schemata. In the end, missing headers lead to abundant tabular data being shrouded and inaccessible to many data-driven applications.

In this chapter, we introduce a fully automated, multi-phase system that discovers table column headers for cases where headers are missing, meaningless, or unrepresentative for the column values. We refer by *schema* to an ordered set of non-empty column header-strings of a table. Our approach is to leverage existing table headers from *web tables* to suggest human-understandable, representative, and consistent headers for any target table. We evaluate our system on web tables that have been extracted from Wikipedia. Overall, 60% of the *automatically* discovered schemata are *exact* and *complete.* Considering more schema candidates, top-5 for example, increases this percentage to 72%. A paper based on the content of this chapter is under submission.

This work makes the following contributions:

1. **End-to-end schema discovery.** A schema discovery system that automatically extracts meaningful column headers for given relational tables based on a corpus of (web) tables.
2. **Efficient similarity search.** A similarity search algorithm optimized for matching web table columns.
3. **Unbiased column similarity measure.** A similarity measure for web table columns that efficiently matches similar value sets regardless of their length.

4. **Context-aware column header assessment.** A schema evaluation approach that finds the overall optimal label composition for a table given multiple sets of column candidate headers.

5. **Evaluation.** A systematic, empirical evaluation of the effectiveness of our schema discovery system on hundreds of real-world web tables.

This chapter is organized as follows. First, we motivate and formally define the problem of schema discovery in Section 3.1. Then, we discuss related work in Section 3.2. We give some background information on similarity search in Section 3.3 and coherence measures in topic modeling in Section 3.4. We describe our schema discovery system and its components in Section 3.5. The results of the empirical evaluation are discussed afterwards in Section 3.6. Finally, we conclude in Section 3.7.

3.1 Missing schema: Dark data

A table is a two-dimensional matrix of data values that are stored either in *machine-readable formats*, such as a relational table in a database or as a structured document (JSON, CSV, XML, etc.) in a file system, or in a *human-readable formats*, such as embedded tables in a PDF file, a plain text file, or in an HTML web page. Tables usually contain a high density of information that exceeds their core content to also provide their structure and, with that, extra semantic meaning.

A *relational table* is a special format of tables that consists of one header row, the schema, which indicates the domains of the values in the data rows, and a *relational instance*, which is the set of rows, i.e., records that hold the actual data values [Codd, 1970]. Ideally, the schema provides a specification of the table that enables simple understandability by humans and integrability by machines.

Definition 3.1. *Given a relational table τ consisting of n columns, we formally define S_τ, the* schema *of τ, as an ordered set of non-empty header-strings $S_\tau = \{h_1, ..., h_n\}$. Each element h_i is a header of the column c_i.*

A table's schema is among the most widely used types of metadata in many scenarios. The majority of traditional schema matching approaches make use of column headers to find a mapping between two data sources [Bernstein et al., 2011]. The main subject for query answering in table search engines are column headers (combined with the table's context and body). For instance, Pimplikar and Sarawagi [2012] emphasized that the completeness and the descriptiveness of the table's schema directly impacts the search results. The existence and the quality of a schema also significantly impacts the process of extending a table with additional attributes [Lehmberg et al., 2015].

Several systems have been introduced to match web tables to a knowledge base (KB) in order to enrich them [Dong et al., 2014; Efthymiou et al., 2017; Ritze et al., 2015]. The existence and quality of headers strongly influence these matching efforts.

Nevertheless, the semantic embedding-based approach by Chen et al. [2019], is able to predict the KB class, independently of the availability of table headers. Systems, such

as [Fernandez et al., 2018a,b], acknowledge that missing schemata aggravate the data discovery problem and suggest similarity-based methods to semantically group datasets.

The lack of a schema limits the abilities to understand, explore and query a table. Furthermore, the quality of the table's schema has a direct impact on the performance of any later application. A schema of good quality is complete, descriptive and does not contain any meaningless, unreliable or opaque column names (and values). Because tables with missing or low quality header are hardly usable for many downstream tasks, we refer to them as *Dark Data*.

Nowadays, there is a wealth of tables in online published documents both in machine and human-oriented formats. The *WebTables* project was the first to automatically detect and extract human-readable tables from the Web [Cafarella et al., 2018]. Their corpus has estimated 154 million diverse HTML relational-like tables. Lehmberg et al. [2016] also extracted a collection of 51 million relational tables from the HTML Web Common Crawl. Open Data is another flavor of online published tables in machine-readable format. Open Data consists of databases published by federal governments, companies, and academic institutions as part of the Open Data movement. Nargesian et al. [2018] crawled 215 393 CSV tables from the UK, US and Canadian Open Data Portal.

In principle, tables on the Web can be viewed as a tremendous distributed database that covers a wide variety of topics, which makes them an appealing source of information. But schemata have shown to be missing in a non-negligible amount of tables on the Web. The majority of relational web tables have, however, missing or only obscure schemata [Balakrishnan et al., 2015; Cafarella et al., 2008]. In [Embley et al., 2006; Wang et al., 2012; Yakout et al., 2012] the authors found that web tables are often lacking schemata or their headers are ambiguous, non-informative, or very generic. Open Data tables seldomly have informative or reliable schemata [Nargesian et al., 2018].

In this chapter, we tackle the problem of discovering a schema for a table with a missing or a low quality schema. Solving this problem sheds light on the dark data and makes it accessible to downstream applications. Our key intuition is to leverage the share of web tables *with* schemata to discover a suitable, coherent, and human-understandable schema for any user-specified table with missing schema. We refer to the task of discovering suitable, coherent, and human understandable schemata for tables with missing headers as *the schema discovery problem*.

Definition 3.2. *Given a table τ with missing or meaningless headers and a corpus of tables $\mathcal{T}$ with headers, the* schema discovery problem *is to find a schema S_τ with meaningful and coherent headers taken from the headers of $\mathcal{T}$.*

Figure 3.1 shows an overview of our schema discovery system. In the first step, similarity search, it automatically discovers for each column in the input table a set of similar columns (with a header) in a web table corpus. Then, it constructs candidate schemata by systematically combining the discovered headers from each individual column with the possible headers of the other columns. In the second step, coherence check, our system measures how strong the individual headers in the final schema fit together based

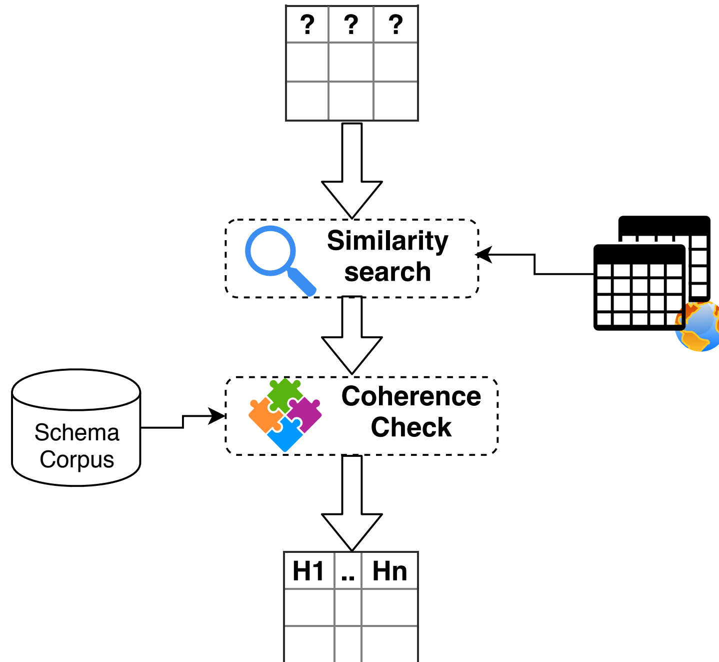


Figure 3.1: Overview of our system

on their co-occurrence in a schema corpus. In other words, we measure how coherent the schema is and use this measure to rank the schema candidates. The schema discovery system finally outputs the top-ranked schema(ta).

Although our system could infer column headers from any table corpus with informative headers, we use web tables extracted from Wikipedia pages (*Wiki web tables*) as our main source of knowledge, because their schemata are, in general, of high-quality. Tables from Wikipedia also cover a wide range of topics, allowing our approach to produce header candidates without introducing any additional semantic data, such as querying a Knowledge Base. Although Wiki tables are only a fraction of all web tables, Bhagavatula et al. [2013] pointed out that these high-quality Wiki web tables are sufficiently valuable to enable table search; hence, they are also a good foundation for our use case.

In order to solve the schema discovery problem, we need to overcome three main challenges: efficient similarity search, unbiased similarity calculation, and coherent schema discovery. We now motivate these three challenges step by step.

First, web tables have a special shape: They are usually small compared to traditional relational tables or to Open Data tables. They have on average five columns and around ten rows per table [Miller, 2018]. Still, the range for the number of columns and rows is surprisingly large (see Figure 3.4 on page 59 for the case of Wiki web tables). Despite their relatively small sizes, exact similarity search in web tables is computationally expensive, memory exhausting and disk straining, because the corpus is huge. A similarity check between each column in the input table and all columns in Wiki web tables ($\sim$170k tables) is unaffordable. A possible solution is to build inverted indexes that map every distinct value of the web tables to a list of columns that contain it. Web tables contain many

distinct values. For example, there are $\sim 185m$ unique values in web tables extracted from the Web Common Crawl [Zhu et al., 2019]. Consequently, an inverted index-based solution is infeasible for web table corpora as it results in huge indexes that make memory management an issue. Thus, we implemented an efficient approximate similarity search approach that reduces memory consumption.

In addition, a similarity measure is needed that matches similar columns independent of their cardinalities, because the number of distinct values often differs significantly in different columns. Statistically, the cardinalities of Web table columns follow a power-law distribution; thus, the column cardinalities have a skewed distribution [Zhu et al., 2016]. In our header discovery scenario, this means that an input column could have a large cardinality difference to the web table column with an optimal header. Therefore, the similarity measure must not be biased towards column cardinalities.

Finally, the discovered schema should consist only of header candidates that appear naturally together. For example, *"Color"* and *"Medal"* are possible headers to a column with the values {*Silver, Gold, Bronze*}. But, once this column co-occurs in a schema that has another column with the header *"Team"*, it is preferable to choose *"Medal"* over *"Color"*. So, we need to design a schema quality measure to quantify and, in this way, eventually improve the overall quality of the proposed schemata.

Schema discovery problem formalization

We interpret the *schema discovery* problem as a combination of a *similarity search* problem and a *coherence optimization* problem: The first part of our solution is a similarity search to find good candidate headers for each individual input column given a table corpus $\mathcal{T}$ – here a corpus of Wiki web tables. More specifically, the similarity search task is described as follows: Considering each column of table τ as a set of distinct values, find the set Y_i of top-k most similar columns for each column c_i in τ from all tables t in $\mathcal{T}$ with the following properties:

(a) Each similar column in Y_i has a header.

(b) Y_i columns are only taken from tables t that have a table-level similarity $sim_t(\tau, t)$ within some threshold θ_t.

(c) Y_i consists of the top-k most similar columns to c_i in terms of a column-level similarity measure sim_c.

First, assuring property (a) is important, because not all columns in the tables of $\mathcal{T}$ are guaranteed to offer a header. Second, because a table represents the context for its columns, we demand in property (b) that not only the columns need to be similar, but also their entire table (with a threshold on table similarity). Applying property (b) as a filter sorts out similar columns from other domains. Third, we propose a top-k filter for the number of similar columns in property (c) for both performance and quality reasons (see Section 3.5).

The second part of our schema discovery solution is an optimization process that composes the desired schema from the candidate headers, aiming for the best coherence

score. A *coherence score* $C(S_\tau)$ of a schema S_τ is a measure of how closely the headers h_i within the schema S_τ are related to each other. We present a concrete measure to quantify this quality aspect of a schema, based on co-occurrence in other schemata, in Section 3.4. We use the headers of columns in Y_i as candidates headers H_i to construct a schema S_τ as the combination in $\times_{i \in 1 \ldots n} H_i$ that maximizes coherence $C(S_\tau)$.

3.2 Related work

Our approach is based on several related works that we describe throughout the chapter, mainly in Sections 3.3 and 3.4. Here, we discuss two related approaches that tackle an overall similar problem to our schema discovery problem.

The *WebTables* project was an effort to create a corpus of "high-quality relational web tables" from Google's gigantic general-purpose Web Crawl [Cafarella et al., 2008]. While conducting the project, Cafarella et al. ran into the problem that a large portion of the extracted web tables had no schema. They tried to generate syntactic schemata for every header-less table by matching the content of each column to a database of 6.8M tuples that covered 849 different domains. However, according to their own analysis, *"we found extremely few tables with clean enough string data to match our controlled database"*. This mismatch inspired us to leverage the labeled part of the (wiki) web tables to generate headers, i.e., schemata for the non-labeled part of the web tables, because the labeled part is expected to have the same structure and properties as the non-labeled part, and it also covers a wide variety of domains. As stated in Section 3.4, this work also suggested a schema coherence score based on PMI, which is one of the 15 coherence score implementations that we tested.

Another approach to find meaningful headers for web tables without a schema was made by Wang et al. [2012]. Their main goal was to understand the concepts that each web table describes, which should then be reflected in the assigned schema. Their efforts consequently enable semantic search functionality that returns relevant tables to an issued keyword query. The header generation phase of the proposed algorithm follows a similar roadmap as our schema discovery system with three major differences: (1) It uses the Microsoft ProBase knowledge base as its main source of information, whereas we use web tables; (2) it uses the top-k concepts as header candidates for each column via direct knowledge base lookup and not via similarity search; (3) it generates the final schema as the combination of column concepts with the highest confidence score returned by the ProBase Knowledge API and not our coherence score.

The proposed schema generation approach, as a pre-processing step for understanding web tables, also follows a different goal than our approach and, hence, produces incomparable results: The algorithm tries to find the combination of column-*concepts* that most likely represents the table-*concept*, whereas we try to find the combination of existing column-*headers* that (according to the reference data) best fit the columns' values and have a high co-occurrence probability. Our approach does not condition the schema to represent any (real-world) entity or concept, because this approach tends to produce less specific and, in particular, different headers than existing tables. The head-

ers we discover are previously used headers that also tend to appear together in the schema corpus; they are not general attribute labels of concepts.

In the following sections, we present the foundations of similarity search and topic modeling, and discuss how they are used to solve our problem.

3.3 Similarity search

We model the problem of finding a single missing header as a set similarity search problem: Given a column without a header, we search for similar columns with headers in the very large search space of all web table columns. The similarity search needs to be efficient enough to process large web table corpora, but it should prioritize the quality of the results (headers) over query time.

This section is divided into three parts. First, we present the similarity measures that we use in our approach and discuss existing solutions to estimate them with limited storage. Then, we present state of the art solutions for set similarity search and discuss whether they fit our problem. We finally describe the data sketching technique that we apply in our schema discovery system.

3.3.1 Set similarity estimation

A set similarity function $sim(A, B)$ maps two value sets A and B to a similarity value in [0,1]. This number measures the similarity between A and B. If A and B are identical, i.e., both contain the same values, $sim(A, B) = 1$; if A and B are completely different, i.e., the two value sets share no value, $sim(A, B) = 0$; for any other value overlap, $sim(A, B)$ should yield values in between 0 and 1. Many set similarity functions exist, and we refer to [Choi et al., 2010] for an overview.

In our scenario, we need to calculate $sim(A, B)$ for a very large number of column- and table-pairs. For this reason, the techniques to estimate the similarity of two tables and two columns, respectively, need to be not only accurate but also efficient. Because column value sets and tables value sets have different characteristics w.r.t. their similarity calculation, we propose two different functions, which we denote as sim_c and sim_t, respectively. The table-level similarity $sim_t(A, B)$ measures the similarity between a header-less input table and a web table, where A and B are the value sets of these two tables. The column-level similarity $sim_c(A, B)$ measures the similarity between a column without header from the input table and a candidate column with header from the web table corpora, where A and B are the value sets of these two columns. In our solution, we use two well-known similarity functions, namely Jaccard similarity as sim_t and Jaccard containment as sim_c [Jaccard, 1901].

Jaccard similarity

Jaccard similarity is the size of the intersection of two sets divided by the size of their union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

We use Jaccard similarity as sim_t to guarantee that the table's columns have at least some common content with the input table; we filter out such web tables that are dissimilar.

Estimation. Because the exact similarity check is too expensive for large numbers of sets, Broder introduced the MinHash technique to efficiently estimate Jaccard similarities with high accuracy [Broder, 1997]. Calculating MinHash sketches, though, requires the hashing of every item in the set with k hash functions. This is computationally expensive, because k is usually in the range between hundreds to thousands, if an estimation of high accuracy is required [Shrivastava and Li, 2014]. Hashing techniques, such as OOPH [Shrivastava, 2017], have emerged to reduce this cost at the expense of a noticeable decrease in the quality of the similarity estimation. The Lazo index method introduces a cardinality-based heuristic to improve OOPH's estimation quality [Fernandez et al., 2019].

Our system, however, uses set cardinalities to estimate sim_t with HyperLogLog (HLL) data sketches [Flajolet et al., 2008; Heule et al., 2013] (see Section 3.5.3).

Jaccard containment

Jaccard containment, also known as *inclusion coefficient*, is an asymmetric measure that captures the ratio of the intersection size of two sets to the size of one of the sets:

$$C(A,B) = \frac{|A \cap B|}{|A|} \tag{3.2}$$

Jaccard containment is preferable over Jaccard similarity for measuring the similarity between sets that have a high variance in their cardinalities. This is true for web table columns, because their distinct value counts statistically follow a power-law distribution [Zhu et al., 2016]. Thus, we use Jaccard containment as sim_c to find relevant columns where A and B consist of the values of two columns. We also experimentally tested the use of Jaccard containment as sim_t, but it filtered out too many tables. Hence, we use Jaccard similarity as sim_t.

Estimation. Because the exact calculation of containment between a large number of column pairs is a performance issue, we need to approximate this calculation as well. Zhang et al. [2010] used bottom-k sketches to estimate the Jaccard similarity, which then serve to estimate containment as follows: $C(A,B) = \frac{J(A,B)}{J(A \cup B, A)}$. By translating a Jaccard containment threshold into a Jaccard similarity threshold, Zhu et al. [2016] estimate containment using MinHash. The Lazo index introduces a method to estimate both

Jaccard similarity and containment simultaneously from MinHash sketches [Fernandez et al., 2019]. Nazi et al. [2018] introduced BML – a maximum likelihood estimator to estimate containment between two sets represented by HLL sketches.

Our system also uses HLL to estimate sim_c, but we do not introduce a direct containment estimator like BML. As our experiments in Section 3.6.3 show, our method for containment estimation solves the problem exactly, i.e., as accurate as BML even without the estimator overhead.

3.3.2 State-of-the-art similarity search

Finding the closest neighbor to a given query point from a set of n points within a d-dimensional space is a widely studied problem and well-known as *nearest neighbor search*. A generalization of this problem is *similarity search*: It describes the distance of items by their similarity and, then, finds closest similar items, i.e., neighbors.

Many real-world applications require a solution for this problem. In information retrieval, for instance, similarity search serves to answer search queries with a ranked list of top similar documents [Haveliwala et al., 2002] and it helps to detect near-duplicate web pages for search result filtering [Broder et al., 1997]. In database research, similarity search is used in *similarity joins* that try to find join-able tables [Augsten and Böhlen, 2013; Zhu et al., 2019]. Similarity search is one of the core components of data discovery systems such as Armur [Fernandez et al., 2018a,b] and Goods [Halevy et al., 2016]. Finally, in data mining, collaborative filtering depends on similarity search to match users by similar tastes, which is necessary to make recommendations [Shi et al., 2014].

There are two variations of similarity search within a d-dimensional space: one that finds all items with a similarity above a given threshold and one that finds the top-k most similar items to a given item. Because the search space for the similarity search problem grows exponentially with the number of dimensions d, finding the *exact* answer to a similarity search query results in exponential query times and/or space requirements. This problem is known as the *dimensional curse* [Andoni and Razenshteyn, 2015]. In our scenario, the search space consists of all columns in the web table corpora that includes $25k$ tables with width between 1 to 400 as shown in Figure 3.4b (page 59).

Some approaches focus on accelerating the similarity search without sacrificing the exactness of the results using indexing and optimization strategies [Arasu et al., 2006; Bayardo et al., 2007; Xiao et al., 2011; Zhu et al., 2019]. Conversely, various approximation methods have been proposed for the similarity search problem that either transform the input data to a lower dimensional representation and/or approximate the similarity measure computation [Wang et al., 2014].

Locality-sensitive hashing (LSH) is the state-of-the-art approximate solution for similarity search. It reduces the complexity from $O(n^2)$ (all pairs comparison) to $O(n)$ [Indyk and Motwani, 1998]. LSH indexes the data items with a hash table in a way that similar items collide and, hence, fall into the same bucket. Thus, any item that collides with the hash of the input item is a candidate. The hashing is an approximation step, but it achieves an effective dimensionality reduction.

Based on the similarity measure, various flavors of LSH indexes have been developed for different similarity measures: Jaccard similarity (MinHash LSH) [Indyk and Motwani, 1998], Euclidean distance [Datar et al., 2004], Hamming distance [Indyk and Motwani, 1998], Cosine distance [Charikar, 2002], Jaccard containment (LSH Ensemble) [Zhu et al., 2016], and coupled Jaccard similarity and Jaccard containment (Lazo index) [Fernandez et al., 2019].

These approaches are all possible alternatives for implementing the similarity search phase in our approach. However, they work well only for a certain similarity measure (except Lazo) and our similarity search solution supports top-k queries using *any* similarity measure that can be defined in terms of set cardinalities. Furthermore, they require powerful compute environments to optimize the query time and, in this way, cope with large table corpora. Our solution in contrast offers high quality results with low main memory requirements. We compare experimentally against Lazo as a representative approximation method in Section 3.6.4.

3.3.3 HyperLogLog (HLL)

As mentioned in Chapter 2, data sketching techniques serve to represent data in sufficiently compact forms and to significantly speed up calculations, e.g., of similarity scores. HLL sketches in particular approximate set cardinalities and, therefore, constitute a well-fitting solution for our scenario: They help to overcome the dimensionality curse of similarity search in web tables and enable the calculation of both Jaccard similarity and containment.

HLL [Flajolet et al., 2008; Heule et al., 2013] is a data sketch that provides a near-optimal estimation for the number of distinct values in a given set (i.e., the set cardinality). HLL is popular and adapted even in databases engines, such as PostgreSQL. One HLL sketch requires approximately 1kB of memory regardless of the underlying set size. We introduced HLL in Chapter 2, we now only review the main aspects of this data sketching technique.

Construction. The input items of a set (i.e., the values of a column) are hashed and the hash value is divided into prefix p and suffix q. Prefix p is used as an index into an array of registers. Each register contains the maximum leading zero count among all suffixes q that have been mapped into it.

Estimators. The maximum leading zero count of each register can be used to estimate the cardinality of the entire set. Register-level estimates are then averaged (by harmonic mean) and corrected to obtain an overall cardinality estimate for the set.

Estimation Error. HLL maintains a relative estimation error < 0.01 for sets with up to 10^9 distinct values [Harmouch and Naumann, 2017] (as we experimentally showed in Chapter 2).

Set operations. HLL sketches support set union operations in a lossless fashion. The result of combining two HLL sketches is exactly the same as if the union of the

two sets had been fed into one HLL sketch. Note that this operation works only if the two input HLL sketches used the same hash functions and the same number of buckets. Because the intersection operation is not natively supported by most HLL implementations, we summarize possible methods of estimating the set intersection and explain which one we choose in the next subsection.

Set intersection estimation

The *intersection* between the two sets A and B has the cardinality $O = |A \cap B|$. The larger the intersection is, the higher is the probability that A and B are samples from the same population [Rice, 2006]. Finding the exact value of O requires comparing each element in A to each element in B.

The set intersection can be calculated either using the *sieve, (inclusion-exclusion) principle*:

$$|A \cap B| = |A| + |B| - |A \cup B| \tag{3.3}$$

or based on the *Jaccard coefficient* [Beyer et al., 2007; Dasu et al., 2002]:

$$|A \cap B| = J(A, B) \cdot |A \cup B| \tag{3.4}$$

$$|A \cap B| = \frac{J(A, B)}{J(A, B) + 1} \cdot (|A| + |B|) \tag{3.5}$$

Estimation. As we experimentally showed in Chapter 2, an estimation of the cardinalities $|A|$, $|B|$ and $|A \cup B|$ can be obtained efficiently and with low standard error using HLL. If an estimation of the Jaccard similarity is also available (using, for instance, MinHash), one can use Equations (3.4) and (3.5).

MinHash is biased towards smaller sets and we observe a huge variance in set cardinalities in our web tables scenario. Hence, to avoid adding additional data sketches, we estimate the set intersections by Equation (3.3) in our similarity search phase, which uses cardinalities estimated by HLL sketches.

In summary, we use Jaccard similarity for table-level similarity sim_t and Jaccard containment for column-level similarity sim_c in the similarity search step of our solution for the schema discovery problem. We need to calculate sim_t and sim_c for a very large number of table- and column- pairs. For this reason, We estimate both similarities using HLL as the data sketching technique.

Given a column without a header, similarity search step finds a set of most similar columns with headers in all web table columns. For every column in the header-less input table, a set of header candidates is extracted. Next, we need to suggest a schema for the input table consists only of header candidates that appear naturally together. Thus, we discuss in the next section methods to quantify how good a set of words (in our case headers) fit together.

3.4 Topic coherence

In the past, research efforts from different disciplines focused on measuring the strength of how facts within a set support each other; more specifically, they aim to quantify how *coherent* they are. The term *schema coherence* debuted in the context of auto-completion of a schema for database designers based on word co-occurrence statistics for headers in schemata, extracted from web tables [Cafarella et al., 2008]. Furthermore, text mining approaches automatically learn a set of important words from unlabeled documents to model a specific topic; these approaches also use coherence measures to support the interpretability of their models [Newman et al., 2010].

In order to develop a suitable coherence score $C(S_\tau)$ for the evaluation of our discovered schema candidates, we adopt measures from topic modelling by treating schema headers as word sets that model a topic.

Given a word set W, Röder et al. [2015] represent a topic coherence measure as a composition of four steps: The coherence score is calculated by splitting W into word pairs (*segmentation*), then calculating the co-occurrence probability of each pair (*probability calculation*), then translating the probabilities into a measure of agreement for each pair (*confirmation*), and finally combining all confirmation measures into one final coherence score (*aggregation*). In our case, the word set W is the schema candidate S_τ.

1. *Segmentation* (Sg) is the division of W into a set of subset pairs. Each subset contains one or multiple words from W. A pair of subsets is formed by pairing every individual subset to, for example, every other subset, only the proceeding subset, or only the succeeding subset.

2. *Probability calculation* (P) is the method to derive word probabilities from the reference corpus. The word probability is estimated by the proportion of documents that it appears in with respect to all documents in the reference corpus. Other variations make use of formatting information and estimate at sentence, paragraph, or sliding window level.

3. *Confirmation measure* (m) assesses the agreement of each subset pair and how strong they support each other. It is computed based on the probabilities corresponding to the subset pair.

4. *Aggregation* (σ) is the function used to combine all confirmation measures of all word pairs into one score $C(S_\tau)$.

We need to choose carefully each part to design our coherence measure $C(S_\tau)$ to fit our problem.

We evaluated three single-word-based *segmentation* methods, i.e., methods that recognize each subset as only one header, namely: Sg_{one} pairs each header with every other header in the schema, Sg_{pre} pairs each header with only proceeding headers and Sg_{suc} pairs each header with only succeeding headers.

The reference corpus for our coherence measure is the schema corpus. For the schema corpus, we propose to use the attribute correlation statistics database (ACSDb) [Cafarella

et al., 2008], because it is the richest available corpus of schemata with statistics. ACSDb lists each unique schema as a set of headers along with a count that indicates how many relations contain it. This directly affects our option for *probability calculation* and limits it to document level, i.e., the header either exists in a schema in ACSDb or not. For the other three parts, we tried the best variations of methods reported by Röder et al. [2015], measured their effectiveness, and used the best-performing ones.

Because we calculate the probabilities based on ACSDb and not on the Wiki web table corpus, as discussed earlier, from which we extract the headers, there is a chance of encountering a column header or a pair of column headers that never appeared (in this combination) in the reference corpus. For this reason, our system implements a smoothing step. More specifically, we add a small constant ($\varepsilon = 10^{-12}$) as recommended by Stevens et al. [2012] to all probabilities.

We consider five different *confirmation measures* in our experiments. Given a pair of headers (h, h'), the confirmation measure m of this pair can be computed as either conditional m_c [Mimno et al., 2011], difference m_d [Douven and Meijs, 2007], Fitelson m_f [Fitelson, 2003], Log-ratio m_{lr} or normalized log-ratio measure m_{nlr}. The latter two measures are well known as *point-wise mutual information* (PMI) and *normalized point-wise mutual information* (NPMI) [Bouma, 2009], respectively. In [Cafarella et al., 2008], the schema coherence score is computed with PMI.

If $P(h)$ is the probability of the header h and $P(h, h')$, $P(h|h')$ are the joint and conditional probabilities of h and h' to occur in the same schema, the exact measure definitions are as follows:

$$m_c(h, h') = \frac{P(h, h')}{P(h')} \tag{3.6}$$

$$m_d(h, h') = P(h|h') - P(h) \tag{3.7}$$

$$m_f(h, h') = \frac{P(h|h') - P(h|\neg h')}{P(h|h') + P(h|\neg h')} \tag{3.8}$$

$$m_{lr}(h, h') = \log \frac{P(h, h') + \varepsilon}{P(h) \cdot P(h')} \tag{3.9}$$

$$m_{nlr}(h, h') = \frac{m_{lr}(h, h')}{-\log(P(h, h') + \varepsilon)} \tag{3.10}$$

Finally, popular *aggregation* functions are minimum, maximum, arithmetic mean and median, and geometric mean. We choose the arithmetic mean σ_a, because Röder et al. [2015] showed that it produces a score that best correlates with human ratings.

As we show in the evaluation section, the best combination of coherence measure parts is:

$$C(S_\tau) = (Sg_{suc}, m_c, \sigma_a) \tag{3.11}$$

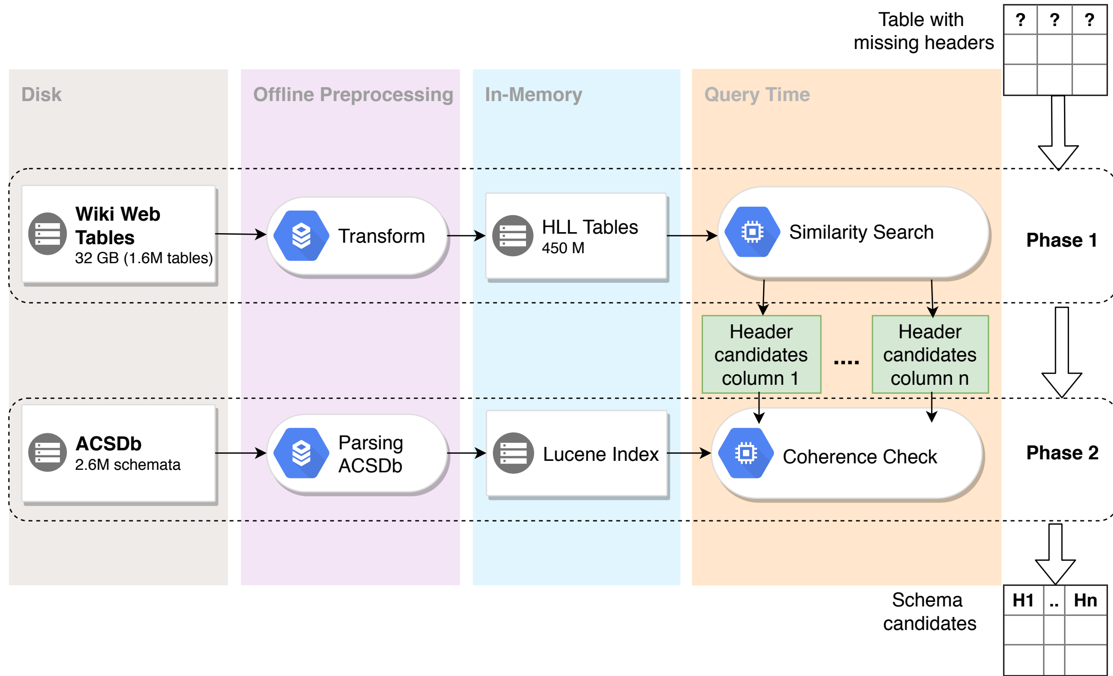


Figure 3.2: Overview of our multi-phase system for coherent schema discovery from Wiki web tables

3.5 Missing schema discovery

After providing the essential foundations, this section focuses on the workflow of our schema discovery system. An overview of the multi-phase missing schema discovery system is shown in Figure 3.2 on page 52.

We start by describing the data structures that are used to efficiently represent the input data. We then discuss the offline pre-processing that constructs these data structures from the two given corpora – the web and the schema corpus. Afterwards, we describe in detail the similarity search for candidate headers and finally the assembly of candidate headers into a coherent schema.

3.5.1 Data structures

Our schema discovery system uses two main data structures: HyperLogLog-based table representations (short *HLL tables*) and an inverted index on schemata statistics (short a *schema index*). The HLL table is a data structure to represent each table in the table corpus with a limited amount of memory and still guarantee an efficient estimation of both sim_t and sim_c (details in Section 3.5.3).

The mapping between a table and a HLL table is shown in Figure 3.3. Each HLL table is an array of *HLL columns*, one for each attribute in the original table. Each HLL column is a pair of a header string and a HyperLogLog data sketch constructed from the set of distinct values of the original data column. Converting all tables from the wiki

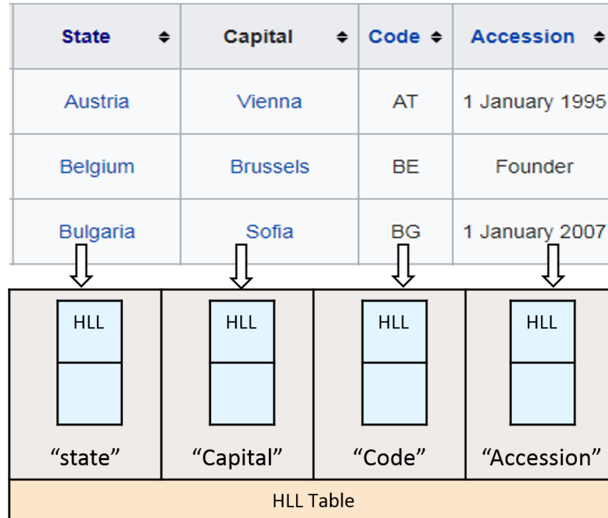


Figure 3.3: HLL table representation of a Wiki web table

web table corpus into this data structure reduces the size of the table corpus from 32 *GB* to 450 *MB*, which fits easily into the main memory of modern computers.

Given that HLL sketches naturally support union set operations, a HLL sketch of the values of the entire table is exactly the result of the union of its column HLL sketches. We later show that the similarity search can effectively be conducted using only the HLL table representation of the table corpus.

To measure the coherence of a schema, we need to query the schema corpus. The computation of the probability of a given header to appear in a schema and the computation of the probability of two headers to co-occur in some schema are two expensive operations, because they need to retrieve all schemata that contain a certain header. To accelerate these retrieval steps, we build a disk-based inverted index, the schema index, on top of the schema corpus, pointing each column header to all the schemata it occurs in.

Both data structures, the HLL tables and the schema index, are built once in an offline step, i.e., before the system is queried for schemata. In the following, we list the steps to prepare and build the two structures.

3.5.2 Offline pre-processing

Tables without a header in the web table corpus are worthless for our system, because they do not provide column header candidates – we remove all such tables from the table corpus. Then, we scan each table once and transform it into a HLL table. During the transformation, empty columns are also ignored, because they lack the basis for a similarity comparison in the similarity search phase. Adding such columns to the HLL tables would consume unnecessary extra space for the respective HLL sketches and they would generate random header candidates for any empty column in the input tables, i.e., the tables for which our system is supposed to find meaningful column headers.

The ACSDb schema corpus contains 2 219 785 distinct schemata, together with their occurrence frequencies. The frequency refers to the number of tables in the relational tables of the Google Web Crawl that have the same set of headers. To build the schema index, we parse the ACSDb corpus and build a Lucene inverted index on top: Each schema is inserted z times into the index, where z is the frequency of this schema as provided by the ACSDb corpus, to ensure the propagation of the schema frequencies to the co-occurrence probabilities of its headers. Before inserting a schema into the schema index, we execute several preparation operations, such as trimming and lower-casing each header and removing special characters including {`.;:!?,`}.

3.5.3 Phase 1: Similarity Search

To construct a coherent schema candidate for a table with missing schema, we first search in the table corpus for similar columns w.r.t. the columns' values. Algorithm 1 shows the similarity search process in detail. As Figure 3.2 on page 52 depicts, the *similarity search* phase expects the HLL table corpus H and a table τ with missing schema as input. The only required disk access is to read and load these two files into memory.

Apart from the HLL tables and the header-less query table, two additional parameters are required: k and θ_t. The parameter k is the maximum number of header candidates to keep for each column in τ; the parameter θ_t is a similarity threshold for the table similarity measure sim_t.

Note that we provide only the essential parameters in Algorithm 1; additional parameters for tuning the HLL structures are provided in the evaluation section.

Because our system tries to find not only reasonable column headers but coherent schemata, it first searches for overall similar tables and then within these tables for similar headers. The threshold θ_t defines which tables are considered to be similar. The intuition is that similar columns have higher probability to occur within a similar table t than in arbitrary tables; hence, we consider only those tables t for the column similarity search that are at least θ_t similar to τ w.r.t. the similarity sim_t between t and $h\tau$. Our experimental results confirm this intuition in Section 3.6.6.

The array of priority queues (*candidates*) is a local data structure that stores the output of the similarity search. The header candidates for each column $c_i \in \{c_1, ..., c_n\}$ in τ are stored in a priority queue *candidates*$[i]$ of size k, which ranks the headers by the similarity between their column and c_i. The algorithm first initializes *candidates* according to n, which is the number of columns in τ, and k, which is the maximum number of candidate headers per column (Lines 1 and 2). The input table τ is then transformed into a HLL table $h\tau$ (Line 3) to enable the calculation of sim_t and sim_c with the tables in the corpus H.

After the first initialization steps, the tables in the table corpus are scanned one after another. Each table t in H is checked for whether its Jaccard similarity $sim_t(t, H\tau)$ to the input table is larger than the threshold θ_t (Lines 7 to 14). The HLL sketches of both t and $h\tau$ are constructed by merging the HLL sketches of their columns.

Algorithm 1 Web tables similarity search

Data: H: the HLL table corpus.

Input : τ: a table with n columns missing headers.
k: the number of candidates for each column.
θ_t: a table-level similarity sim_t threshold

Output: *candidates*: an array of n priority queues (one for each column in τ).

▷ Initialization
1 **for** $f \in \{1, ..., n\}$ **do**
2     $candidates[f] \leftarrow priorityQueue(k)$

▷ HLL table
3 $h\tau \leftarrow table2Hyper(\tau)$
▷ Merge of column HLLs into a table HLL
4 $HLL_{h\tau} \leftarrow merge(h\tau.getColumns())$
5 $card_{h\tau} \leftarrow HLL_{h\tau}.cardinality()$
6 **foreach** *table* $t \in H$ **do**
7     $HLL_t \leftarrow merge(t.getColumns())$
8     $card_t \leftarrow HLL_t.cardinality()$

    ▷ Early pruning of tables
9   **if** $\frac{min(card_t, card_{h\tau})}{max(card_t, card_{h\tau})} < \theta_t$ **then** Break;

    ▷ Inclusion-Exclusion principle
10     $HLL_{all} \leftarrow merge(HLL_t, HLL_{h\tau})$
11     $card_u \leftarrow HLL_{all}.cardinality()$
12     $card_n \leftarrow card_t + card_{h\tau} - card_u$
13     $table_{sim} \leftarrow \frac{card_n}{card_u}$ ▷ Jaccard similarity
14   **if** $table_{sim} \geq \theta_t$ **then**

    ▷ Find top-k most similar columns for each column from the input table
15   **foreach** *column* $c_i \in h\tau$ **do**
16   **foreach** *column* $c_j \in t$ **do**
17             $HLL_{both} \leftarrow merge(c_i, c_j)$
18             $card_{cu} \leftarrow HLL_{both}.cardinality()$
19             $card_{cn} \leftarrow c_i.cardinality() + c_j.cardinality() - card_{cu}$
            ▷ Containment between columns
20             $column_{sim} \leftarrow \frac{card_{cn}}{c_i.cardinality()}$
21             $weight_{sim} \leftarrow column_{sim} \cdot table_{sim}$
22             $candidates[i].insert(c_j.getHeader(), weight_{sim})$

Because estimating the actual Jaccard similarity for the two tables is expensive, we prune some of these estimation steps by checking the ratio between the two table cardinalities (Line 9): Given two sets A and B (i.e., tables in our case) where $|A| \geq |B|$, if $\frac{|B|}{|A|} < \theta_t$ then $J(A, B) < \theta_t$ is also true, because $|A \cap B| \leq |B| \leq |A| \leq |A \cup B|$. Hence,

if $\frac{|B|}{|A|} < \theta_t$ is already true, we can skip the calculation of $J(A, B)$. This optimization step is used also in other work [Bayardo et al., 2007; Xiao et al., 2011].

If t passes this test, the Jaccard similarity is estimated given only the HLL sketches of t and τ; otherwise, the algorithm estimates both the cardinality of each table and the union cardinality from the HLL sketches (Lines 10 to 11). Likewise, the cardinality of the intersection is estimated using the inclusion-exclusion principle from these cardinalities (Line 12). If $sim_t(h\tau,t)$ is above θ_t, i.e., if the overall tables are similar (Lines 13 and 14), the algorithm calculates all pair-wise similarities between the columns of $h\tau$ and t (Lines 15 to 20).

For each column pair (c_i, c_j), where c_i belongs to $h\tau$ and c_j belongs to t, it estimates the containment using the HLL sketches of the columns. The higher sim_c between any column c_j and a column c_i is, the more likely it is that c_j's header also fits c_i; hence, we use c_j's header as a candidate header for c_i if it is among the top-k headers with the largest value overlap. Thus, the denominator of the containment is the cardinality of c_i (Line 20). Again, we use HLL sketches to estimate the intersection as described above.

Finally, the column containment sim_c is weighted by the tables' Jaccard similarity sim_t to increase the rank of a column from a similar table in the respective priority queues (Line 21). Boosting the column similarity with the table similarity is the first step to enhance the coherence of the final schema, because it favors headers from more similar tables once the header candidate is inserted in the respective priority queue (Line 22).

Algorithm complexity. The worst case complexity of Algorithm 1 for n columns in an input table is $\mathcal{O}(|\mathcal{T}| + n \cdot \mathcal{T}_c)$, where $|\mathcal{T}|$ is the number of tables in $\mathcal{T}$ and $\mathcal{T}_c$ is the number of all columns in $\mathcal{T}$. Both are constants for the fixed table corpus $\mathcal{T}$.

The coherence of the final schema is checked again in Phase 2, which we describe in the following section.

3.5.4 Phase 2: Coherence Check

The *coherence check* phase is responsible for building a meaningful and coherent schema from the up to k header candidates that the system calculated in the previous phase for each column. Algorithm 2 shows the coherence check process in detail.

The algorithm calculates a list (*schemata*) containing the top-m possible schemata for a table τ ranked by their coherence measure $C(S_\tau)$. We already discussed the details of the coherence measure calculation in Section 3.4 and it represented by (Lines 3 to 11) in Algorithm 2.

The input of the algorithm consists of the schema index I, the top-k header candidates *candidates* and the desired number of best schema candidates m. The process then starts by initializing the priority queue *schemata* of size m to retain the most coherent possible schemata (Line 1). The local data structure SC stores all possible schemata that can be assembled from the different top-k candidate header lists; it is calculated as the Cartesian product over all n header sets *candidates*[1], . . . , *candidates*[n].

The algorithm scores each schema s in SC (Lines 3 to 11) before inserting it into the priority queue (Line 12). First, a set PS of header pairs is constructed using the

Algorithm 2 Coherence check

Data: I: The Schema Index.

Input : *candidates*: an array of header candidates. (output of Algorithm 1).
m: the number of schema candidates.

Output: *schemata*: Top-m coherent schemata

```
1  schemata ← priorityQueue(m)
   ▷ Schema candidates
2  SC ← X_{i∈1...n} candidates[i]
3  foreach s ∈ SC do
       ▷ Header pairs
4      PS ← segment(s)
5      MC ← Φ
6      foreach (h, h') ∈ PS do
           ▷ Probabilities
7          p ← probability(h', I)
8          pc ← condProbability(h, h', I)
           ▷ Confirmation measure
9          m_c ← pc / p
10         MC.insert(m_c)
       ▷ Schema coherence
11     c ← arithmeticMean(MC)
12     schemata.insert(s, c)
```

$Sg(suc)$ segmentation method that groups each header with the succeeding header in the schema s. Then, the agreement of each pair of headers (h, h') in PS is assessed using the conditional confirmation measure m_c (Line 9).

The measure m_c is the ratio of pc, which is the probability of h and h' to co-occur together in the same schema, and p, which is the probability of seeing the header h' within any schema of I. Thus, the algorithm queries the schema index I to retrieve the required statistics (e.g., number of schemata in which a header appears or in which two headers co-occur) to calculate p and ps (Lines 7 and 8). The arithmetic mean of the set MC of all m_c values of all pairs in PS, then, is the coherence score of the schema s (Line 11).

The schema s is finally inserted into the *schemata* priority queue of size m together with its coherence score. Because *schemata* is bound in size to m, it in the end contains the m most coherent schemata for the header-less input table τ (Line 12).

Algorithm complexity. The complexity of Algorithm 2 for n columns in an input table is $\mathcal{O}(k^n \cdot \frac{n \cdot (n-1)}{2})$, where k^n is $|SC|$ and $\frac{n \cdot (n-1)}{2}$ is the complexity of $segment(s)$; k is constant.

3.6 Experiments

The goals of our experimental evaluation are (1) to evaluate the effectiveness of using HLL sketches for Jaccard similarity estimation between tables and Jaccard containment estimation between columns; (2) to compare several well-known topic coherence measures to rank the discovered schemata; and (3) to evaluate the ability of our approach to discover correct or meaningful schemata for web tables. More specifically, this section supports the following insights:

Section 3.6.3. HLL sketches offer an efficient and accurate estimation method for Jaccard similarity and containment. This is true even with the skewed distribution of column cardinalities in web tables.

Section 3.6.4. The best header of a column can be expected to appear within the top-5 header candidates.

Section 3.6.5. The coherence score that uses Sg_{suc} as segmentation method and m_c as confirmation measure is the optimal configuration for the majority of schemata in the Wiki web table corpus. Thus, we propose it as the default measure of coherence in our system.

Section 3.6.6. Our system produces meaningful, coherent schemata. In our experiments, 60% of the automatically discovered schemata are exact and complete; considering top-5 schemata, this percentage increases to 72%.

Section 3.6.7. Many discovered schemata that do not exactly match the ground truth are still semantically very close (and, hence, good): They differ only in synonyms, plural forms, short forms, or symbolic representations. Thus, the effectiveness scores of our method measured in Section 3.6.6 are even higher when the discovered schemata are evaluated by humans.

3.6.1 Experimental setup

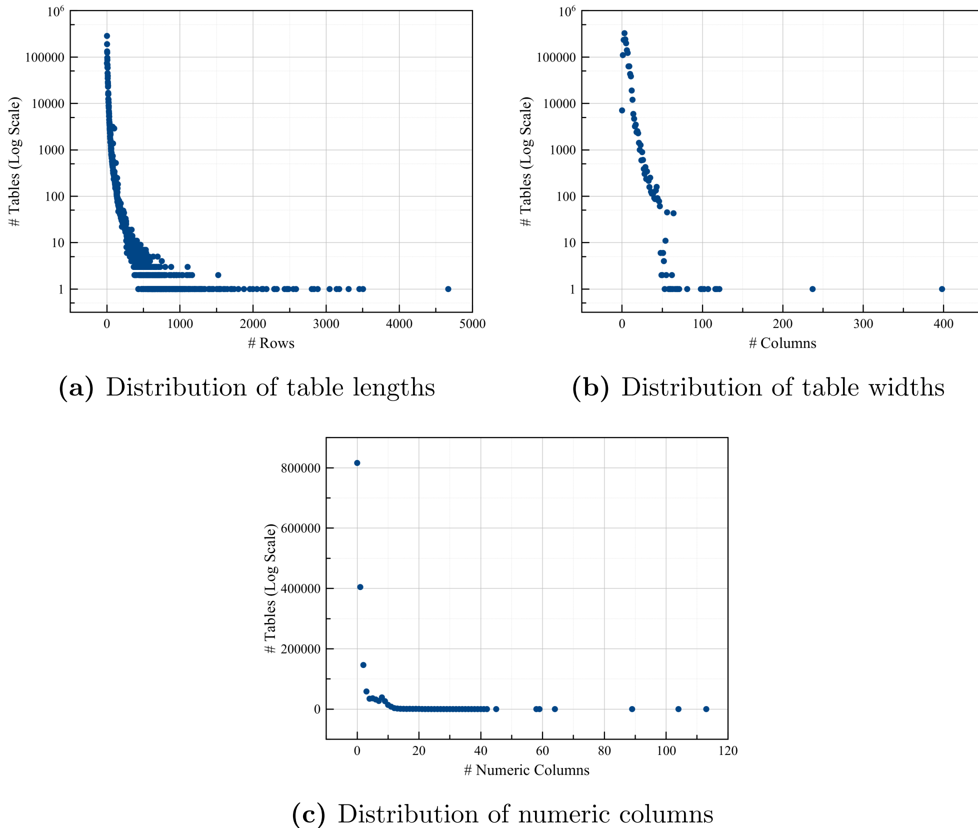
Hardware

We tested our system on a Dell PowerEdge R620. The server has two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) processors, 128 GB DDR3-1600 RAM and runs CentOS 6.10. The implementation is a multi-threaded Java application for Oracle's JDK 64-Bit Server VM 1.8.0_151.

Parameters

Our schema discovery system requires four parameters: the table-similarity threshold θ_t, the number of header candidates for each column k, the number of final schemata in the result m, and the expected standard error of the HLL sketches defined by the length of the prefix p.

The value of θ_t should enable the retrieval of the most similar tables from the table corpus, but not too many tables. A low value of θ_t severely affects the runtime, as it



(a) Distribution of table lengths

(b) Distribution of table widths

(c) Distribution of numeric columns

Figure 3.4: Basic statistics about the Wiki web tables corpus

adds many additional column to be tested for similarity. We experimentally test several values for θ_t in Section 3.6.4 and use $\theta_t = 0.4$ if not stated otherwise, because it provides a good balance between runtime and result quality (for web tables).

We tested $k = 5$ and $k = 10$. Because the runtime and results improved with $k = 5$, we use this value by default. Furthermore, we use $m = 10$ as the default value for the number of output schema candidates, but, in practice, it is up to the user's preference. Our HLL sketch implementation is based on [Heule et al., 2013]. It uses a 64-bit MurmurHash function[1] and is parameterized to provide an estimated standard error of 0.01, i.e., we set a prefix $p = 14$ for the normal representation and $p = 25$ for the sparse representation. For the Lazo index, we use the official Java implementation[2].

Datasets

We use the Wiki web tables corpus [Bhagavatula et al., 2013] as table corpus and the attribute correlation statistics database (ACSDb) [Cafarella et al., 2008] as schema corpus.

The *Wiki web table* corpus has 1.6 million Wikipedia tables in JSON format. They have been extracted from HTML tables with "wikitable" class attribute and were then

[1] `https://sites.google.com/site/murmurhash/`
[2] `https://github.com/mitdbg/lazo`

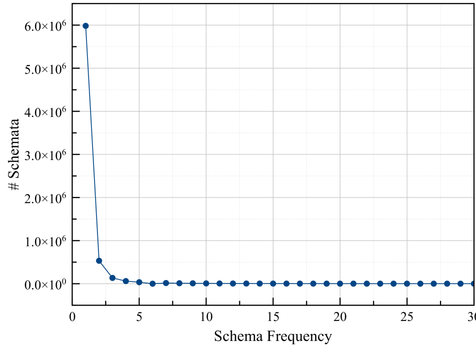


Figure 3.5: Schemata frequency distribution in ACSDb.

normalized into regular matrices of attribute values. Figure 3.4 illustrates the statistical characteristics of this corpus. We use these statistics for the interpretation of our results.

The *ACSDb* corpus stores statistics about schemata used in tables on the Web. This second corpus is needed to compute the probability of seeing a header in a schema, as well as, the conditional probability of a header knowing the presence of another header in the same schema. The ACSDb corpus contains 5.4 million unique headers and 2.6 million unique schemata. Only a relatively small number of schemata in ACSDb appears frequently, while most schemata are actually rare as shown in Figure 3.5.

Quality Measures

Human rankings would serve as the best quality measure to evaluate the results of our schema discovery system, but evaluating the large number of test schemata in our experiments manually is not feasible. Therefore, we judge only a small sample (100 tables) of discovered schemata and 400 headers manually for error analysis and, for the majority of tests, hide the existing original headers from the system, and then test whether we can re-discover them.

We apply a train-test split validation by dividing the Wiki tables into train and test tables using two different settings: In the first setting, we have 5% test tables (24 770 tables) and 95% training tables (1 627 997 tables). In a 70%/30% setting, we split the Wiki tables into 30% test tables (149 163 tables) and 70% training tables (1 503 608 tables).

While the training tables provide header candidates, i.e., they serve as the table corpus, the test tables are used as input for the header discovery. Because both splits show very similar results (quality-wise), we present only the results for the 95%/5% split in the individual charts. We randomly sampled the test tables and used the same split in all experiments. We made sure that there is no intersection between the train and test tables so that a table cannot be used to find its own header. In reality, though, tables can appear multiple times in automatically harvested table corpora.

We use the quality metrics Precision (P), Recall (R) and F-measure ($F1$) to assess the quality of each suggested schema. To calculate them, we compare the original and the suggested schemata pair-wise. Based on the automatic schema matching evaluation

analogy from [Do et al., 2002] we define *true positives* as exactly re-discovered headers; *false positives* as in any degree different headers from the original schema; and *false negatives* as no-header suggestions, i.e., cases where no header could be suggested for an input column.

For example, if the original table schema is {*"year", "population", "growth"*} and the result is {*"year","village population",""*}, then P is $\frac{1}{2}$, because *"year"* is a match and the second attribute is a non-match. Similarly, R is $\frac{1}{3}$, because the column *"growth"* received no header proposal. In Section 3.6.7 we examine close but not exact matches.

3.6.2 Schema discovery with knowledge bases

A large body of research focuses on solving the problem of matching web tables to a KB, such as [Chen et al., 2019; Dong et al., 2014; Efthymiou et al., 2017; Limaye et al., 2010; Ritze et al., 2015]. Any approach that matches a web table to a KB class can be seen as a baseline for our approach, if we interpret the properties of the assigned KB class as a schema proposal. The proposed schema, then, includes only the subset of properties that we are able to map to the columns of the header-less web table.

In general, KB-based approaches are different from ours in that they are able to propose (if at all) column headers for only such tables that match KB classes. They, however, are unable to infer schemata for non-KB concepts, such as sports results, annual reports, or fishing supplies. Our approach, in contrast, is able to match a table to any concept and combination of concepts seen anywhere on the Web, which also covers concepts that are not represented in KBs. As shown by Ritze et al. [2016], only ∼3% of relational web tables describe DBpedia[3] entities and only 562 445 out of 137M (∼32%) columns have a property correspondence. Hence, KB-based approaches cannot derive any header for at least ∼97% of the web tables and miss at least 68% of the columns within the matchable tables.

Because the pool of headers provided by a KB contains only a fraction of possible headers in web tables, KB-based schema discovery approaches are not able to suggest specific headers, such as the *exact* headers of web tables. For example, only 2% of the headers in our test tables are in fact also DBpedia properties. Hence, no DBpedia-based approach can reconstruct them and compete with our approach in suggesting *specific* headers.

3.6.3 Similarity estimation using HLL

HLL sketches offer high data compression, fast construction times (due to using only one hash function), and excellent accuracy in estimating set cardinalities. Nevertheless, there is no quality guarantee for estimating the similarity between two sets of arbitrary values, given only the sets' HLL sketches. Thus, we experimentally tested the efficiency of our two HLL-based similarity estimation methods, i.e., the Jaccard similarity sim_t and the Jaccard containment sim_c estimations (see Section 3.3.1), which both use the HLL table representation of the web tables.

[3] `https://wiki.dbpedia.org/`

For this experiment, we calculated the exact Jaccard similarity for all pairs of test tables and the exact Jaccard containment for all pairs of test table columns. These exact similarity values serve as our ground truth for the estimated similarity values. Before discussing the quality of the estimated similarities, we point out that estimating the similarities instead of exactly calculating them yields an extreme reduction in memory and runtime: The exact calculations took a week of runtime, while our estimation methods consumed only six hours, including building the HLL sketches from the input tables.

Quality-wise, we observed that the HLL-based estimations of sim_t and sim_c are, in fact, *exact* in all tested cases. This is because HLL sketches guarantee near exact cardinality estimations for small value sets [Harmouch and Naumann, 2017; Heule et al., 2013] and the number of values in both columns and tables is low on average in our web table corpus: The average column cardinality is 10, confirming the statistics by Zhu et al. [2019], and the average number of distinct values in a table is 40. The exactness of the cardinality estimations propagates into exact Jaccard similarity and containment estimations. Despite the already small size of the web tables, their HLL table representations are still more than 98% smaller without losing a relevant amount of accuracy to calculate sim_t and sim_c.

3.6.4 Similarity search evaluation

To evaluate the effectiveness of the similarity search phase, we trained our schema discovery system, built a Lazo index on the training split of the web table corpus, and ran the similarity search for all tables in the test split with a fixed memory of 100GB. During the similarity search phase, we recorded the top-10 header candidates for each column ($k = 10$). Given that the Lazo index does not support top-k queries, we set the containment threshold to 0, ordered the results by containment score, and considered only the top-10 header candidates to make it comparable to our results. We ran Lazo with an increasing number of permutations to evaluate the trade off between quality and used resources.

To systematically study the effect of the table similarity threshold θ_t, we repeated the experiment with values $\theta_t \in \{0.0, 0.1, 0.2, 0.3, 0.4\}$. We also omitted the table similarity filtering in an additional experiment, i.e., we ran Algorithm 1 without lines 9 and 14.

Table 3.1 lists for every threshold θ_t the percentage of columns whose original header is in the top-10 proposed headers (we discovered the *exact header*), the percentage of columns whose exact original header is not among the top-10 proposed headers (we discovered a *differing header*), and the percentage of columns where no header was proposed at all (we discovered *no header*). Table 3.1 also shows the average *indexing* time and the average *query* time per column. The same statistics are also reported for the headers suggested by the *Lazo* index method with permutations 1, 8, and 10.

Using a Lazo index, a higher number of permutations causes an increase in both the indexing time and the number of headers that are exactly recovered. Due to the limited memory, the Lazo index fails to index the training tables using 10 permutations (the largest dataset reported by Fernandez et al. [2019] has only 10k columns). Our system consumption remained less than 25GB. Compared to the Lazo index with 8 permutations,

	% exact header	% differing header	% no header	Indexing (sec)	Query (sec)
Lazo (1)	30.95	38.52	30.52	1109	∼0
Lazo (8)	31.63	39.60	28.77	1236	∼0
Lazo (10)	out of memory			-	-
no filtering	62.73	36.16	1.10	1016	0.97
$\theta_t = 0.0$	74.00	24.10	1.10	1016	0.52
$\theta_t = 0.1$	61.66	13.49	24.85	1016	0.32
$\theta_t = 0.2$	42.90	7.92	49.18	1016	0.28
$\theta_t = 0.3$	27.25	4.02	68.73	1016	0.26
$\theta_t = 0.4$	16.77	1.98	81.25	1016	0.26

Table 3.1: Percentage of columns with exact, differing, and no discovered header and runtime (indexing and query per column) w.r.t. different experimental settings.

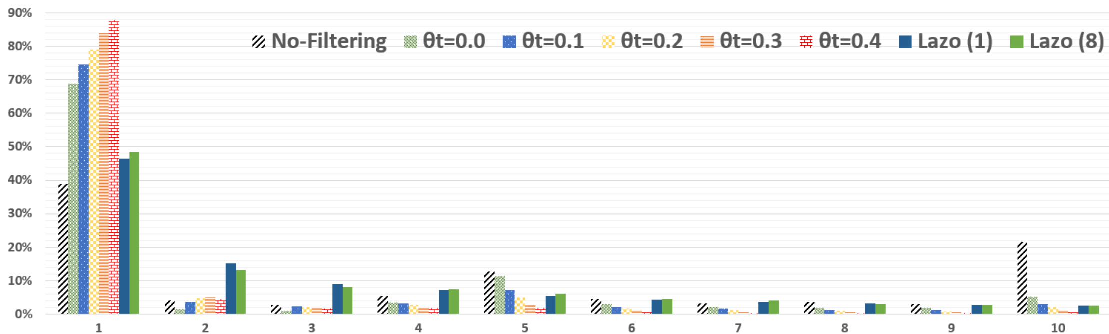


Figure 3.6: Percentage of columns with exact discovered headers at each position in the top-10 header candidates.

our system is capable of proposing twice the number of original headers with fewer incorrect headers and only 1.1% of no headers. With the same differing-header ratio, we also suggest at least twice as many exact headers. The indexing time of our data structures is constant, but Lazo requires much more memory to provide the same quality as our system does.

The higher we set the threshold θ_t, the more tables from the table corpus are filtered out during similarity search, because we require a higher minimum Jaccard similarity between them and the input test table. This trend is reflected in Table 3.1 by the percentage of columns with no discovered header. On the other hand, the percentage of columns for which the exact header appears in the top-10 header candidates is inversely proportional to θ_t, i.e., the likelihood of finding the exact header increases when filtering out fewer tables. Simultaneously, the query runtime per column increases when we filter out less tables, i.e., with lower θ_t.

Our intuition for conducting the table similarity filtering step is that the table is the context for all of its columns. By pre-conditioning the table similarity, we rank headers from similar tables higher. Thus, the correct header is at a higher position in the top-k

header candidates list. To show this effect experimentally, Figure 3.6 presents for each threshold θ_t at which position (1 to 10) in the respective header candidates lists the exact headers occur. When we apply no table similarity filtering, the exact headers, if found, appear at arbitrary, mostly lower positions.

However, with table similarity filtering, the exact headers are ranked higher and appear largely within the top-5 header candidates; even the top-1 header candidate is in most of the cases (69% to 88%) the exact header. A higher threshold θ_t, in general, ranks the exact header at a higher positions. In conclusion, with the table similarity filtering and the similarity threshold θ_t we can force our system to report high-quality headers at higher positions in the header candidates list.

Figure 3.6 also shows that Lazo ranks exact headers in a similar manner to our system but not as good as ours. In general, one can trade the header quality for the number of columns that receive no header recommendations and vice versa (see Table 3.1). If a use case requires headers and can tolerate low quality recommendations, we suggest $\theta_t = 0$; use cases that do require reliable headers should use some $\theta_t \geq 0.4$, because false headers are, then, less than 2%.

The numbers reported in Figure 3.6 also show that a candidate ranking of size $k = 5$ is sufficient to find the exact headers for the majority of the tested columns regardless of θ_t; hence, we use this as the default ranking size.

3.6.5 Coherence measure comparison

Our schema discovery system implements an optimization process that aims for the schema with the maximum coherence score. As discussed in Section 3.4, the coherence score is a combination of four components: a segmentation method (Sg), a confirmation measure (m), a probability calculation (P), and an aggregation function (σ). Having already made a decision for P and σ, we still need to find the optimal combination of implementations for Sg and m.

To find the combination that maximizes the coherence score of the discovered correct schema, we tested 15 implementation combinations (3 segmentation methods $\times$ 5 confirmation measures). Each of the 15 combinations was used to test the usefulness of the coherence score for the schema discovery on a random sample of 1000 test web tables.

Table 3.2 presents for each combination of Sg and m and for schema precisions $\in \{30\%, 50\%, 70\% \text{ and } 100\%\}$ the percentage of tables that achieves at least that precision level with the specific combination. For example, with Sg_{suc} and m_c (first row), our system correctly discovered 30% of each schema for 66.1% of all test tables; with the same combination, it correctly discovered the entire schema for 22.2% of the test tables.

The maximum percentage of tables for which our approach was able to correctly find the complete and exact original schema uses the combination Sg_{suc} as segmentation method and m_c as a confirmation measure. Therefore, we propose to use the coherence measure $C(S_\tau) = (Sg_{suc}, m_c, \sigma_a)$ with a Boolean-document level method for probability calculation. To provide some examples, the top-10 coherent schemata in our web table

Sg	m	Precision (p)			
		30%	50%	70%	100%
Sg_{suc}	m_c	66.1	**54.8**	31.7	**22.2**
	m_f	61.9	49.5	27.4	18.5
	m_d	65.5	53.3	31.2	20.9
	m_{lr}	65.9	52.4	30.4	20.3
	m_{nlr}	65.9	53.9	31.2	20.6
Sg_{pre}	m_c	61.4	48.5	26.7	19.2
	m_f	65.4	53.6	**32.2**	21.9
	m_d	59.3	47.4	26.1	17.8
	m_{lr}	65.2	52.0	30.1	20.2
	m_{nlr}	66.1	53.9	30.7	20.4
Sg_{one}	m_c	64.1	51.3	30.2	20.7
	m_f	64.2	52.2	30.2	20.8
	m_d	63.2	50.8	28.9	18.9
	m_{lr}	65.5	51.9	29.9	19.8
	m_{nlr}	**66.3**	53.8	31.4	20.7

Table 3.2: Precision of schemata proposed for 1000 random test tables w.r.t. 15 combinations of Sg and m.

corpus are listed in Table 3.3.

	Schema	$C(S_\tau)$
1	{mon, tue, wed, thu, fri, sat, sun}	0.97252
2	{friday, saturday, sunday}	0.97078
3	{males, females}	0.94508
4	{females, males}	0.92715
5	{girls, boys}	0.92715
6	{high, low}	0.92625
7	{male,female}	0.92319
8	{gold, silver, bronze}	0.91984
9	{9th, 10th, 11th, 12th}	0.91678
10	{singular, plural}	0.91429

Table 3.3: The top-5 most coherent schemata in our tables corpus

3.6.6 Overall system evaluation

Having demonstrated the effectiveness of the similarity search phase and investigated reasonable settings for the coherence check phase, the following experiment evaluates the overall effectiveness and efficiency of our multi-phase schema discovery system. To this end, we ran both phases using the entire training set and the entire test set, respectively.

	P	R	$F1$	Schema Suggested
$\theta_t = 0.0$	36%	36%	36%	99%
$\theta_t = 0.1$	52%	38%	44%	84%
$\theta_t = 0.2$	62%	30%	41%	61%
$\theta_t = 0.3$	71%	22%	33%	38%
$\theta_t = 0.4$	78%	14%	24%	22%
$\theta_t = 0.4$ (No-weight)	76%	14%	23%	22%

Table 3.4: Micro-average of P, R, $F1$ percentages of the top-1 schemata with varying table similarity threshold.

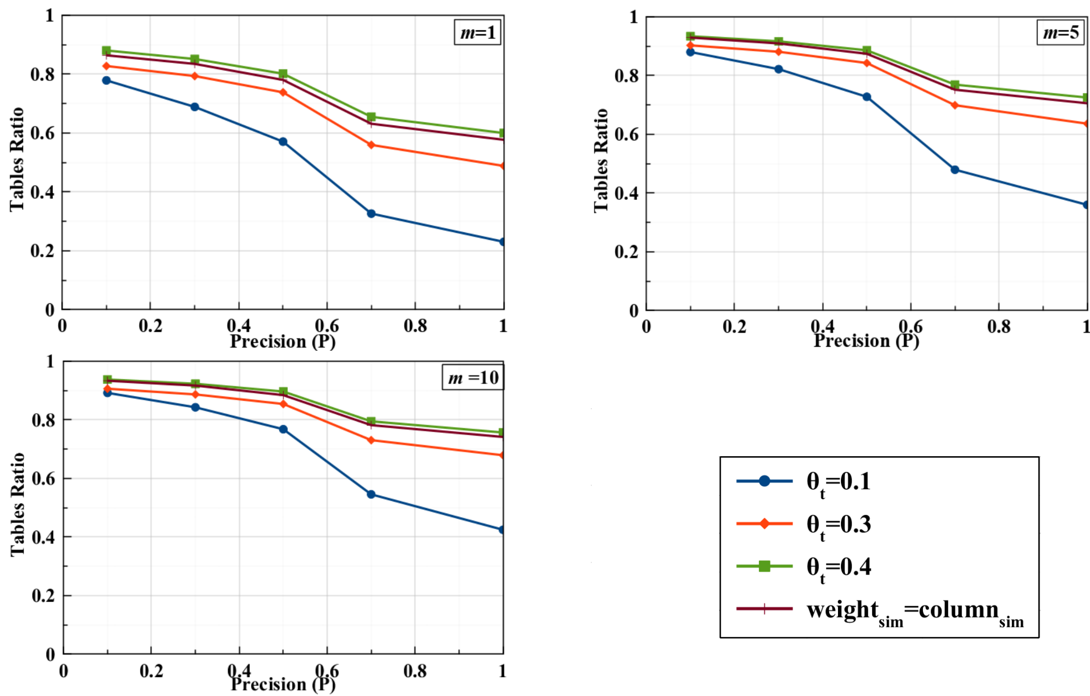


Figure 3.7: Ratio of test tables with proposed schema of precision $\geq P$ w.r.t. varying schema candidate ranking sizes m and similarity thresholds θ_t.

In Section 3.6.4, we concluded that our system ranks high-quality headers, which are the original headers, higher using higher thresholds θ_t, but with many columns receiving no header recommendations as a trade-off. To better understand the effect of θ_t on the *entire* final schemata, we let the discovery system produce the top-m schemata for each table in the test split of the web table corpus with various values $\theta_t \in \{0.1, 0.2, 0.3, 0.4\}$. In Table 3.4, we report on the percentage of the 24 770 test tables for which our system *automatically* suggested a schema ($m = 1$) and the quality of these schemata. With $\theta_t = 0.1$, only 16% of the test tables received no schema suggestion, i.e., the system could not suggest a header for any of the schema's attributes. A higher threshold θ_t results in a higher percentage of tables getting no schema candidates.

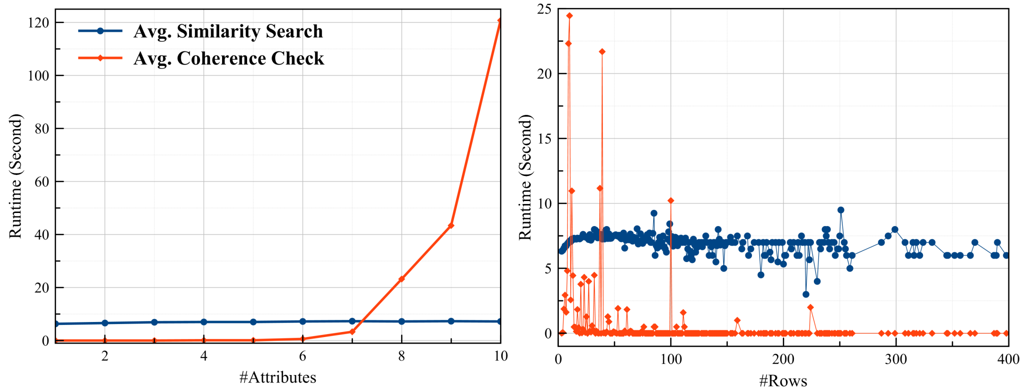


Figure 3.8: Runtimes of similarity search and coherence check phases w.r.t the number of attributes and rows within test tables

Quality-wise, the experiment clearly shows the inverse relationship between precision and recall. Because our system aims to produce meaningful and coherent schemata for header-less tables, it is more important to produce precise schemata (high precision) rather than finding a schema for every input table (high recall). Trading off recall for precision, the output of our system has a higher assurance to be a part of any further data science effort. Next, we quantify the precision levels of the resulting schemata.

Figure 3.7 plots for three schema ranking sizes $m \in \{1, 5, 10\}$ the resulting table percentages for increasing precision, i.e., the percentage of resulting test tables that have P or more correct headers. The left figure shows the results for the top-1 schema candidates, which are the truly *automatically* discovered schemata – they do not require a user to pick a schema from a ranked list of schema candidates. With $\theta_t = 0.4$, 60% of the discovered top-1 schemata are completely and exactly the original schema ($P = 1$). This percentage decreases to 23% with $\theta_t = 0.1$, because most of the schemata that the system finds with the lower threshold do not match the original headers. Therefore, higher thresholds θ_t increase precision as we observe in Table 3.4.

Considering more schema candidates, i.e., larger ranking sizes m, increases the percentage of tables for which our system is able to discover the entire, exactly correct schema to 75% of the cases. The threshold θ_t still controls the quality of the proposed schemata trading off precision of the recommendation and the number of tables with no results. As default setting, we have chosen $\theta_t = 0.4$ to produce schema suggestions of high quality.

In addition, we validated the effect of enhancing the coherence of the final schema by boosting the column similarity with the table similarity (see Algorithm 1, Line 21). For this purpose, we experimentally disabled the step of boosting sim_c with sim_t in Line 21 of Algorithms 1 by removing the $Table_{sim}$ factor from $weight_{sim} = column_{sim} \cdot Table_sim$. We then ran the experiment with $\theta_t = 0.4$ again. Figure 3.7 shows that, without boosting, the quality of the suggested schemata drops, regardless of the size of correctly suggested part of the schema and the number of considered schema candidates. This shows that enhancing sim_c with sim_t improves the precision, especially for $m = 1$ (Table 3.4).

To evaluate the runtime performance of our system, we measured the runtime of each phase per test table. On average, our schema discovery system needs 6.9, 4.2, and 11.1 seconds per table for the similarity search phase, the coherence check phase, and the overall discovery, respectively.

In an additional experiment, we investigated the scalability of our approach w.r.t. the width (#columns) and the length (#rows) of the header-less input table. The similarity search shows a linear runtime behavior with the input table's length and width (see Section 3.5.3). The runtime of the coherence check, in contrast, increases exponentially w.r.t. the input table's width and linear with table's length (see Section 3.5.4). The exponential runtime increase with the input table's width is expected due to the increase of the size of the Cartesian product in Algorithm 2 (Line 2).

3.6.7 Error analysis

To automate our previous evaluations, we demanded the system to produce the *exact* original header as a candidate to count it as a hit; otherwise, it was counted as a miss, although the proposed schemata might have been very similar or even better than the original schema. We now take a closer look at the cases where our system failed to re-discover the exact schema of a test table using the training tables. For this purpose, we manually evaluated 100 non-matching schemata.

During manual investigation, we noticed many cases where non-matching attribute labels actually differed only in plural form, synonym, time, representation, or granularity. Our system explicitly performs lower-casing but it avoids all other transformations, such as stemming and plural removal, because these operations can be error-prone. Table 3.5 presents some of the proposed schemata that are not exact but still meaningful. The candidate headers provided by our system for some test tables simply include *plural forms* of the original headers, such as “champions” instead of “champion”. In other cases, the suggested header is a *synonym* of the original one, such as suggesting “male” as a header for a column with original header “men”, “boy”, or “man”.

Furthermore, many Wikipedia tables represent the same object or concept but at different points in time. Because *time* is often reflected in the column headers, some proposed headers are basically correct but with the wrong time-affix. For instance, the header “season” appears with a year attached to it and we find “2008 season” as a candidate for a column with original header “2009 season”.

We also observe several *representations* of the same header that are interchangeable, such as symbols “#” or short forms like “no.” that both refer to “number”. Another interesting case is when the header candidate is more specific or *granular* compared to the original header. For example, “village population” is a correct header for a column with the original header “population”.

In total, we manually evaluated 400 random headers considered as differing in the experiment with $\theta = 0.4$ shown in Table 3.1. Around 50% of them are actually incorrect headers, mainly due to different date or date range; most of them were proposed for numerical columns. The other 50% headers are differing due to granularity, word form,

Original schema Top-1 proposed schema	Error source
{Year, **Champion**} {year, **champions**}	Plural
{men's, women's} {male, female}	Synonym
{team, city, state, stadium, capacity, **2008 season**} {team, city, state, stadium, capacity, **2009 season**}	Time
{**no.**, date, score, opponent, record} {**number**, date, score, opponent, record}	Short form
{Place, Rider, **Number**, Country, Machine, Points, Wins} {place, rider, #, country, machine, points, wins}	Symbol
{year, **population**} {year, **village population**}	Granularity

Table 3.5: Example cases where our system suggested a different but still very meaningful schema

Error source	% of headers
Incorrect	50%
Granularity	24%
Word form	6%
Spacing	6%
Synonym	12%
Shorter form	2%

Table 3.6: The results of manual evaluation of 400 random headers considered as differing in the experiment with $\theta = 0.4$ shown in Table 3.1

spacing, being a synonym, or being a shorter form of the original header. The detailed results are shown in Table 3.6.

3.7 Summary

In this chapter, we focused on column header as one of the most relevant single column metadata in the context of relational data. We proposed a system to solve the problem of discovering suitable, coherent, and human understandable schemata for tables with missing or meaningless headers.

Our system consists of two phases: similarity search and coherence check. In the first phase, we use a table corpus to discover similar columns for each column in a header-less input table; the headers of the discovered columns serve as a header candidates for the input table's columns. In the second phase, we built schema candidates with the header candidates of all columns; a coherence measure is, then, used to score the schema candidates and keep the most coherent schemata as an output for the user.

We evaluated our approach on a table corpus extracted from Wikipedia pages and a schema corpus extracted from the Google search Web Crawl. Our experiments showed that our system can automatically recover the exact and complete schema for 60% of the discovered schemata; this increases to 75% if we consider the top-10 schema candidates. The similarity search phase of our system is capable of proposing twice the number of original headers with fewer incorrect headers compared to Lazo index. A higher table similarity threshold θ_t, in general, forces our system to report high-quality headers: when some $\theta_t \geq 0.4$ is used, the percentage of false headers is less than 2%.

Through manual inspection, we also found that many discovered schemata that do not match their original schemata exactly are still very similar to their original schemata, because they differ only in synonyms, plural forms, short forms, or symbolic representations.

Chapter 4

The Impact of Missing Values on FD Discovery

High data quality has a significant impact on any downstream data-driven application [De Veaux and Hand, 2005; Haug et al., 2011]. Missing values represent one of the most common and challenging data quality issues. Single-column data profiling tools consider the *number of missing values* in a column as key metadata to indicate data quality. In this chapter, we investigate the impact of this metadata on another data profiling task, namely functional dependencies (FDs) discovery. Functional dependencies also play an important role in maintaining data quality as they can be used to enforce data consistency and to guide repairs over a database.

When using existing FD discovery algorithms, some FDs could not be detected precisely due to missing values or some FDs can be discovered even though they are caused by missing values with a certain NULL semantics. A *genuine* FD is an FD that would be valid if the dataset contained no missing values and no other errors.

We define a notion of genuineness and propose algorithms to compute the genuineness score of a discovered FD. This can be used to identify the set of genuine FDs among the set of all valid dependencies that hold on the data. We evaluated the quality of our method over various real-world and semi-synthetic datasets with extensive experiments. The results show that our method performs well for relatively large FD sets and is able to accurately capture genuine FDs. Our published work in [Berti-Equille et al., 2018] is the basis of this chapter.

Despite its importance, no previous work has focused on these critical aspects of FD discovery over incomplete data. We make the following contributions.

1. We formally and experimentally show the phenomenon caused by missing values over FD discovery.
2. We formalize the definitions of *genuine*, *ghost*, and *fake* FDs and study their impact under various NULL semantics and imputation strategies.
3. We propose a probabilistic approach for estimating the genuineness score of FDs, provide an efficient method for enumerating and pruning irrelevant possible worlds,

and propose an efficient sampling-based algorithm to approximate the probabilistic genuineness score.

4. We propose likelihood-based approaches to efficiently approximate the genuineness score of discovered FDs.

5. We perform an extensive set of experiments of our methods on real-world and semi-synthetic datasets that show the effectiveness and efficiency of our approach.

The rest of the chapter is organized as follows. In Section 4.1, we introduce the problem of FD discovery over incomplete data. Section 4.2 presents related work. In Section 4.3, we provide an illustrative example to motivate the need for this study and then formally define the notions of *fake*, *ghost*, and *genuine* FDs. In Sections 4.5 and 4.6, we propose probabilistic and likelihood-based approaches to quantify the genuineness of an FD. In Section 4.7, we present our experimental evaluation, and finally, Section 4.8 concludes our contribution and suggests future work.

4.1 FDs and incomplete data: Trust

Functional dependencies (FDs) are one of the most important types of integrity constraints and have been extensively studied by the research community [Caruccio et al., 2016; Liu et al., 2012]. FDs have a number of applications, such as maintaining data quality [Fan and Geerts, 2012], capturing schema semantics [Fan et al., 2008], schema normalization [Papenbrock and Naumann, 2017], data integration [Wang et al., 2009], repairing of data inconsistencies, and data cleaning [Beskales et al., 2010; Bohannon et al., 2007].

Let R be a relation with schema $\{A_1, \ldots, A_m\}$ with n tuples and m attributes. The domain of an attribute A_i is denoted as $dom(A_i)$. Let $X \subseteq R$ be a subset of attributes. The projection of a tuple t to a set of attributes X is denoted as $t[X]$.

Definition 4.1. Functional dependency (FD). *An FD $X \to Y$ over a set of attributes $X, Y \subseteq R$ states that X functionally determines Y. X is the determinant (`LHS`) and Y is the dependent (`RHS`). The FD is said to hold on R when $\forall t_i, t_j \in R$, if $t_i[X] = t_j[X]$ then $t_i[Y] = t_j[Y]$. The FD is violated when there exists at least one pair of tuples (t_i, t_j) such that $t_i[X] = t_j[X]$ but $t_i[Y] \neq t_j[Y]$.*

An FD $X \to Y$ is said to be *minimal* if no subset of X determines Y. In other words, removing any attribute from X renders the FD invalid. An FD is said to be *non-trivial* if $X \cap Y = \emptyset$. An FD is said to be *normalized* if the `RHS` is a single attribute. In this study, we consider only the set of minimal, non-trivial, and normalized FDs as they can be used to infer all other FDs that hold on R using Armstrong's axioms [Armstrong, 1974].

Traditional FDs are typically defined for correct and complete data and there are many efficient algorithms to discover FDs from a given clean dataset [Papenbrock et al.,

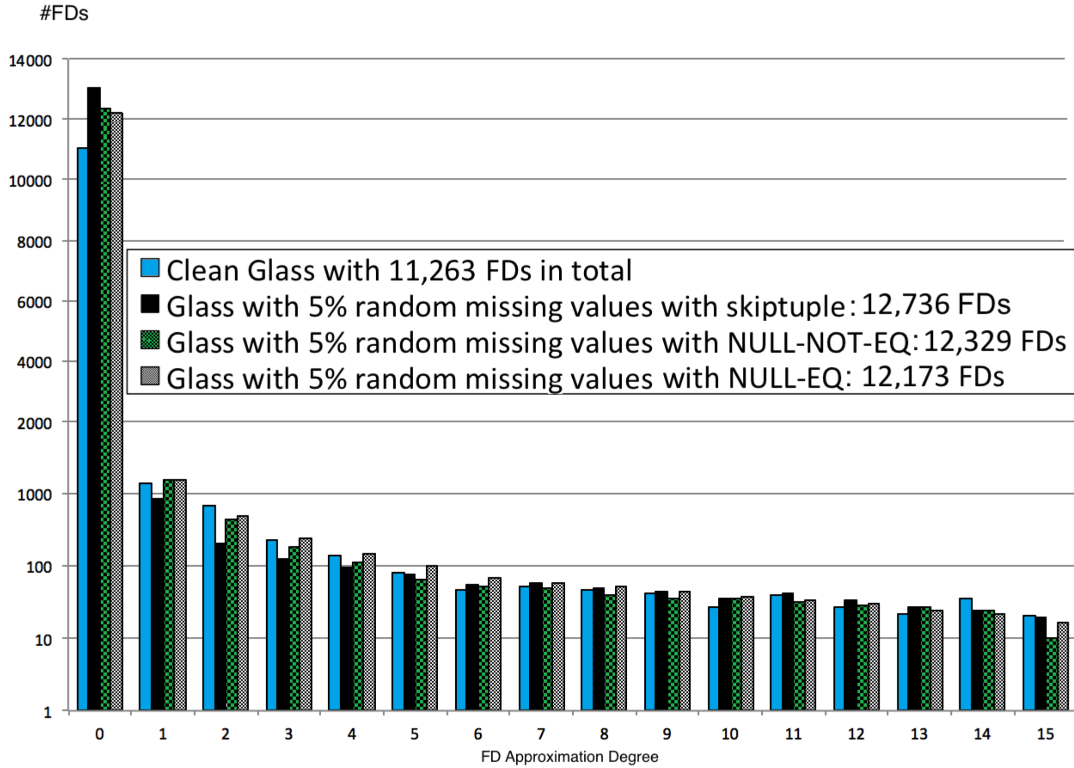


Figure 4.1: Number of FDs discovered from the clean version of Glass dataset and a polluted version.

2015b]. However, many real-world datasets are neither correct nor complete, possibly due to the integration of multiple relations.

In addition, a common implicit assumption is that the proportion of incomplete data is assumed to be relatively low and with no significant impact on the set of discovered dependencies. But it is well-known that data error rates may vary from 20% up to 80% in many real-world datasets [DemandGen, 2017; Gartner, 2007] with dramatic consequences and significant costs [De Veaux and Hand, 2005; Haug et al., 2011].

Thus, traditional FDs often have trouble with incomplete data that routinely exist in massive datasets. The missing values are represented as NULL values, which we denote with $\perp$.

4.1.1 Strategies to handle missing values

Not surprisingly, the database community has come up with a number of workarounds and strategies to handle the issue of missing values. We provide a representative list of such strategies and briefly mention why they are not satisfactory. The experimental results are based on the Glass dataset, a benchmark dataset with 10 attributes and only 214 tuples, that we used as one of the datasets in our experiments (see Section 4.7).

Strategy 1: Skipping tuples with NULLs.

The simplest strategy is to ignore the set of tuples with NULLs and use the remaining subset of the relation to discover FDs. This approach suffers from two problems. First, as mentioned above, large parts of the dataset even entire columns can contain NULL values. Second, this approach also discovers a number of spurious FDs that do not hold on the entire relation.

We illustrate this issue in Figure 4.1 (bars corresponding to approximation degree 0). There are 11,263 FDs on the correct and complete version of Glass dataset. However, when we injected missing values randomly into 5% of the tuples and skipped those incomplete tuples, the number of discovered FDs jumped to 12,736 (on average over 10 runs).

Strategy 2: NULL Semantics.

An alternative approach is to propose definitions of FDs over relations with NULLs. The two commonly used NULL semantics are, `NULL-EQ`, denoted ($\perp = \perp$) or, `NULL-NOT-EQ`, denoted ($\perp \neq \perp$) that treat all missing values as identical or distinct, respectively.

The two semantics have diverse motivations and lead to the discovery of different sets of functional dependencies. If two tuples t_i and t_j have NULL for an attribute A_k, then `NULL-EQ` assumes that both tuples have the same indeterminate value. `NULL-NOT-EQ` assumes that t_i and t_j have different but still indeterminate value.

For example, consider the dataset $R_3(A, B, C)$ (Figure 4.2 on page 78) and the two tuples, t_2 and t_3. Both tuples have NULL for the attribute B. No exact FD holds for `NULL-EQ`. But, two exact FDs, $B \rightarrow A$ and $B \rightarrow C$, hold given the NULL semantic `NULL-NOT-EQ`.

However, as we shall show in Section 4.3, this approach also does not prevent the discovery of spurious FDs. This can also be seen in Figure 4.1 where there are 11,263 FDs on the clean dataset but almost 12,329 and 12,173 FDs for `NULL-NOT-EQ` and `NULL-EQ` respectively.

Strategy 3: Approximate FDs.

Another strategy is to relax the requirement that the FD holds on all tuples. Instead, one can aim to discover *approximate* (a.k.a. partial) or *relaxed* FDs. Approximate FDs are violated by some fraction of the tuples. Relaxed FDs allow attribute values to be similar and do not require them to be equal when checking for violations [Caruccio et al., 2016].

Definition 4.2. Approximate functional dependency (AFD). *An FD is called approximate (or partial) functional dependency if it holds only on a subset of the tuples. We represent an AFD with an approximation degree of α as $X \rightarrow^{\alpha} A$. The approximation degree α can be quantified through the notion of satisfaction error [Kivinen and Mannila, 1995; Liu et al., 2012]. Note that $X \rightarrow^{0} A$ is the same as $X \rightarrow A$.*

The G_3 metric measures the number of violating tuples that must be deleted from R such that the AFD holds exactly (no violation). We use G_3 as the approximation degree measure for the rest of the chapter.

This approach also suffers from multiple issues whereby identifying a suitable threshold is not straightforward and it does not prevent from discovering a large number of spurious FDs.

This is illustrated in Figure 4.1 where the number of approximate FDs discovered over the complete dataset and the incomplete dataset are different for various approximation degrees. For example, there are 1,156 FDs that are violated by exactly 1 tuple (approximation degree 1). However, there are 379, 1328, and 1404 FDs with an approximation degree of $\alpha = 1$ when we skip incomplete tuples or apply `NULL-EQ`, `NULL-NOT-EQ` semantics, respectively.

Strategy 4: Data Imputation.

Data imputation refers to the process of replacing missing data with substituted values [Efron, 1994]. One can use probabilistic or other statistical imputation techniques to fill the missing data, such as [Fan et al., 2012; Song et al., 2015], and run FD discovery on the imputed data. However, the FDs discovered are tightly tethered to the imputation strategy. Further, in the case of probabilistic imputation, it can happen that the FDs discovered are valid only in a small fraction of all possible imputed worlds. Our proposed solutions are based on the concept of data imputation as we discuss in Section 4.5.

4.1.2 The genuineness of FDs discovered over NULLs

We investigated the effect of incomplete data on the discovery of functional dependencies and compare the FDs that are discovered from the dirty dataset and its corresponding clean version using several approaches to handle missing values. It is clear that the FD discovery result from the incomplete data includes spurious FDs that do not hold on the clean dataset and also misses some FDs that do hold.

In order to systematically study this phenomenon, we define three types of FDs: *genuine, ghost*, and *fake* FDs. A *genuine* FD is an FD that would be valid if the dataset contained no missing values and no other errors. When the data is incomplete, a traditional FD discovery could discover false positive (*fake*) FDs that do not hold on the complete version of data, or miss discovering some true FDs (*ghost*) that actually hold on the complete data.

Most current FD discovery techniques do not provide any guarantee regarding the genuineness of the discovered dependencies. Further, these methods neither detect nor remove fake FDs and they do not consider ghost FDs. Note that we do *not* address the quite different problem of judging whether a valid FD is in fact semantically correct. This latter decision can, in principle, be made only by a human expert.

As mentioned earlier, FDs have been used in a number of applications, such as data cleaning and query optimization. Use of non-genuine FDs for such scenarios could have a deleterious effect. For example, ghost FDs could be considered as missed opportunities for schema normalization and data cleaning. On the other hand, fake FDs could cause issues when they are used in query optimization. When used as data integrity constraints, fake FDs prevent the insertion of valid tuples.

In the rest of this chapter, we define the notion of genuineness of FDs and develop algorithms to estimate it. We would like to note that from the perspective of FD discovery, the impact of NULL values and the impact of other erroneous values, such as typos and outliers, are very similar. For example, one could use an orthogonal mechanism to identify typos or outliers and simply set those erroneous cells to NULL that have to be fixed later. However, in the rest of this thesis, we consider only the case of missing values in FD discovery. Regardless, our method can be naturally extended to handle outliers and typos.

4.2 Related work

There has been a number of different formalisms to extend FDs to handle erroneous data inherent in real-world applications. Common approaches include approximate FDs [Bleifuss et al., 2016] and conditional FDs [Huhtala et al., 1998; Kivinen and Mannila, 1995] whereby a FD holds on a subset of data instead of the entire dataset. We refer to [Caruccio et al., 2016] for a detail survey of relaxed definitions of FDs. There has been some work on probabilistic FDs that might hold on the data with some probability [De and Kambhampati, 2010; Wang et al., 2009]. Recently, there has been some proposals to extend the semantics of FDs under NULL markers [Badia and Lemire, 2015, 2017].

However, none of the FD discovery algorithms questions whether the discovered FDs are genuine FDs or not. The usual working assumption is that FD discovery operates from a clean dataset. As an unfortunate consequence, existing FD discovery-based frameworks for data cleaning rely on the correctness of the discovered dependencies; cleaning rules based on matching dependencies [Bertossi, 2011] and constant or variable CFDs [Fan and Geerts, 2012] may actually be erroneous (fake) and skip relevant dependencies (ghost).

There has been some research considering the effect of NULL values on constraints, namely on FDs [Levene and Loizou, 1998] and on keys [Köhler et al., 2016a]. As in our work, the authors first acknowledge the presence of NULL values in typical datasets and explain their detrimental effects on enforcing constraints. They then introduce the notions of possible and certain FDs/keys (weak and strong FDs in [Levene and Loizou, 1998]). A possible FD/key is one for which a possible world exists (*i.e.*, some instantiation of all NULL values with any non-NULL values). A certain FD/key is one that holds for all possible worlds. Both works then construct sound and complete axiom systems for such dependencies and Köhler et al. [2016a] suggest an algorithm for the discovery of certain keys.

Köhler et al. [2016b] go a step further, by proposing an algorithm to discover approximate certain keys, *i.e.*, keys with NULL values that are still sufficient to identify tuples (certain), but may have some violating values (approximate). In a similar vein, certain FDs (with some violations) might be good candidates for genuine FDs. In contrast, we are interested in the behavior of FDs under changing cleanliness to then determine genuine FDs.

Finally, some attempts have been made to solve the problem of inconsistency between data and their respective set of FDs. Chiang and Miller [2011] developed an algorithm for FD repair and maintenance without overfitting the potentially erroneous data. But they did not consider NULL semantics within their cost model for both data and constraint repairs. Another method to maintain an FD set was proposed in [Mazuran et al., 2016]. This method adds one or more attributes to an FD to repair it instead of changing the data. It estimates to what extent an FD is violated by the data using measures of confidence and goodness of an FD. However, the authors excluded attributes with NULL values from being involved in FDs.

4.3 Genuine, Ghost, and Fake FDs

In this section, we first provide an illustrative example of the impact of missing values on the discovery of FDs. Next, we formalize the definition of *genuine*, *fake*, and *ghost* FDs in the case of exact and approximate FDs.

4.3.1 Illustrative example

Let us consider the relation R_0 with schema $R_0(A, B, C)$ in Figure 4.2. We compute the set of exact and approximate FDs from R_0. Table F_0 (Next to R_0) gives a sample of the FDs discovered, with their corresponding approximation degree α. For non-zero approximation degree, we also list the identifiers of the tuples that need to be changed or removed, using "|" as OR operator and "," as AND operator on the same line of the approximate FD. For instance, in the last line of table F_0, the notation of the AFD $C \rightarrow^2 A$ $\{(t_1|t_2), (t_3|t_4)\}$ means that two tuples (t_1 or t_2) and (t_3 or t_4) have to be removed or updated so that $C \rightarrow A$ becomes valid.

Now, suppose we randomly inject missing values (denoted by $\perp$) in the original dataset R_0 to create three polluted versions of the dataset, denoted R_1, R_2, and R_3 containing one, two, and three missing values, respectively.

We recompute the set of exact and approximate FDs for each dataset version, denoted as F_1, F_2, and for F_3 with the two NULL semantics. In R_1 and R_2, both `NULL-EQ` or `NULL-NOT-EQ` are identical as there is only one missing value per attribute.

Then, we compare the original set of FDs reported in F_0 with the sets of FDs discovered from each polluted version. We can observe some interesting differences in the discovered FD sets. For example, when we compare F_0 to F_1, and F_1 to F_2, the more the number of missing values increases, the more FDs are lost for a fixed approximation degree (e.g., the exact FDs $A \rightarrow C$ and $B \rightarrow C$ disappear from F_1 to F_2), and new FDs appear (e.g., $B \rightarrow A$ appears from F_0 to F_1).

Actually, the original FDs do not completely disappear, they lose one approximation degree and "fade out". For example, $A \rightarrow C$ and $B \rightarrow C$ in F_0 and F_1 become $A \rightarrow^1 C$ and $B \rightarrow^1 C$ in F_2 respectively; another example is $C \rightarrow^1 B$ in F_0, which becomes $C \rightarrow^2 B$ in F_2.

R_0	A	B	C
t_1	0	1	1
t_2	0	1	1
t_3	1	1	1
t_4	1	0	1

F_0: α	FDs discovered from R_0	
0	$A \rightarrow C$	
	$B \rightarrow C$	
1	$A \rightharpoondown^1 B$ $\{(t_3 \vert t_4)\}$	
	$B \rightharpoondown^1 A$ $\{t_3\}$	
	$C \rightharpoondown^1 B$ $\{t_4\}$	
2	$C \rightharpoondown^2 A$ $\{(t_1, t_2) \vert (t_3, t_4)\}$	

R_1	A	B	C
t_1	0	1	1
t_2	0	1	1
t_3	1	$\perp$	1
t_4	1	0	1

F_1: α	FDs discovered from R_1	
0	$A \rightarrow C$	
	$B \rightarrow C$	
	$B \rightarrow A$	(fake)
1	$A \rightharpoondown^1 B$ $\{(t_3 \vert t_4)\}$	
2	$C \rightharpoondown^2 A$ $\{(t_1, t_2) \vert (t_3, t_4)\}$	
	$C \rightharpoondown^2 B$ $\{(t_3, t_4)\}$	(ghost)

R_2	A	B	C
t_1	0	1	1
t_2	0	1	$\perp$
t_3	1	$\perp$	1
t_4	1	0	1

F_2: α	FDs discovered from R_2	
0	$B \rightarrow A$	(fake)
1	$A \rightharpoondown^1 B$ $\{(t_3 \vert t_4)\}$	
	$C \rightharpoondown^1 A$ $\{t_1\}$	(fake)
	$B \rightharpoondown^1 C$ $\{(t_1 \vert t_2)\}$	(ghost)
	$A \rightharpoondown^1 C$ $\{(t_1 \vert t_2)\}$	(ghost)
2	$C \rightharpoondown^2 B$ $\{(t_1, t_3) \vert (t_1, t_4) \vert (t_3, t_4)\}$	(ghost)

R_3	A	B	C
t_1	0	1	1
t_2	0	$\perp$	$\perp$
t_3	1	$\perp$	1
t_4	1	0	1

$F_3^{=}$: α	FDs discovered from R_3	
0	$\emptyset$	
1	$B \rightharpoondown^1 A$ $\{(t_2 \vert t_3)\}$	
	$A \rightharpoondown^1 C$ $\{(t_1 \vert t_2)\}$	(ghost)
	$B \rightharpoondown^1 C$ $\{(t_2 \vert t_3)\}$	(ghost)
	$C \rightharpoondown^1 A$ $\{t_1\}$	(fake)
2	$A \rightharpoondown^2 B$ $\{(t_1 \vert t_2), (t_3 \vert t_4)\}$	(ghost)
	$C \rightharpoondown^2 B$ $\{(t_1, t_3) \vert (t_1, t_4) \vert (t_3, t_4)\}$	(ghost)

$F_3^{\neq}$: α	FDs discovered from R_3	
0	$B \rightarrow A$	(fake)
	$B \rightarrow C$	
1	$A \rightharpoondown^1 C$ $\{(t_1 \vert t_2)\}$	(ghost)
	$C \rightharpoondown^1 A$ $\{t_1\}$	(fake)
2	$A \rightharpoondown^2 B$ $\{(t_1 \vert t_2), (t_3 \vert t_4)\}$	(ghost)
	$C \rightharpoondown^2 B$ $\{(t_1, t_3) \vert (t_1, t_4) \vert (t_3, t_4)\}$	(ghost)

Figure 4.2: R_0 is a relation of binary values for attributes A, B, and C; R_1 is the same relation but with one missing value ($\perp$) randomly injected; similarly, R_2 and R_3 have two and three missing values randomly injected, respectively. Exact and approximate FDs are computed from each relation and reported in a tables next to it where α is the approximation degree. Tables $F_3^{=}$ and $F_3^{\neq}$ report the FDs computed from R_3 for the two NULL semantics: `NULL-EQ` and `NULL-NOT-EQ`, respectively.

In this example we deliberately injected missing values, but an integration scenario with data from multiple, heterogeneous sources may well introduce such missing values, which similarly affect FD discovery. As seen in Figure 4.2, the presence of missing values produces some FDs that were not discovered in R_0.

Another interesting phenomenon is illustrated in Figure 4.2, where three missing values were injected in R_0. In the case where NULL values have the NULL-EQ semantics ($F_3^{=}$), exact FDs are no longer discovered but some FDs appear (e.g., $C \rightarrow^1 A$), even though they were not present in the original FD set F_0 with the same approximation degree.

Interestingly, the two FD sets, $F_3^{=}$ and $F_3^{\neq}$ obtained using the two NULL-EQ and NULL-NOT-EQ semantics are very different for the first two approximation degrees. Which one should be selected? Which FD set is the closest to F_0, the set obtained from the original, clean dataset? What if the three missing values were injected differently? Obviously, removing all tuples with missing values would also lead to another quite different FD set, even further apart from the one obtained from the original dataset.

This phenomenon has neither been identified nor studied by previous work: Either FDs are computed from a supposedly complete and error-free dataset, where records with missing values do not exist or are excluded from the FD discovery process, or the methods assume that one of the two default semantics for handling NULL values is systemically applied.

As illustrated in the example, both working assumptions suffer from the discovery of spurious FDs. Understanding the various ways in which spurious FDs appear is extremely important, as FDs have a number of applications in data management. We investigate precisely this phenomenon by first formally defining the types of FDs discovered over NULL values and then developing a series of algorithms to quantify the genuineness of an FD.

4.3.2 Formalization

For ease of exposition, we first define the notion of genuine, ghost and fake for exact FDs. Consider two versions of the relation R: R_{clean} that is clean/complete and R_{dirty} that is a noisy version of R_{clean} with missing values. Let F_{clean} be the set of FDs discovered over R_{clean} while F_{dirty} is the set of FDs discovered over R_{dirty} under a suitable NULL semantics (such as `skiptuple`, NULL-NOT-EQ, or NULL-EQ).

We can see that the set of FDs is not identical. We partition the set of exact FDs in $F_{clean} \cup F_{dirty}$ into the following groups.

Same FDs: These are exact FDs that are present in both FD sets, *i.e.,* $F_{same} = F_{clean} \cap F_{dirty}$;

Fake FDs: These are exact FDs that are discovered from R_{dirty} but not from R_{clean}, *i.e.,* $F_{fake} = F_{dirty} \setminus F_{clean}$. We consider them as false positive FDs – FDs that could be considered valid but are not;

Ghost FDs: These are exact FDs that are discovered in F_{clean} but "disappeared" in F_{dirty}, *i.e.*, $F_{ghost} = F_{clean} \setminus F_{dirty}$. These are false negative FDs – candidate FDs that are considered non-FDs but are indeed valid FDs from F_{clean};

Genuine FDs: Using the notations above, we can see that genuine exact FDs can be reconstructed as $F_{genuine} = F_{same} \cup F_{ghost}$.

Let us now extend these definitions for approximate FDs. Recall that approximate FDs are associated with an approximation degree that measures how many tuples need to be removed such that an AFD can become an exact FD. A simplistic approach would be to consider all AFDs with an approximation degree less than a certain threshold as equivalent to exact FDs and reuse the prior definition of genuineness for exact FDs. However, finding an appropriate threshold is challenging. We provide a generic definition that takes into account both the degree of cleanliness of data and the degree of approximation.

Given a dataset with $x\%$ of missing values, we denote with $F_{x,y}$ the set of valid FDs with approximation degree of exactly y. For example, $F_{0,0}$ denotes the set of *exact* FDs discovered from the *clean* dataset, and $F_{10,3}$ denotes the set of approximate FDs with degree 3 discovered from the dataset containing 10% missing values. Let $F_{x',y}$ be the set of valid FDs discovered from a version of the dataset ($x' > x$) with more missing values with the same approximation degree y. For notational convenience, we denote both the degree of dirtiness and the corresponding dataset with that degree of dirtiness using x.

Definition 4.3. Same FD set. *Given a fixed approximation degree y, $SAME^{x_0}_{x,y}$ is the set of FDs discovered from a dirty dataset x (with $x > 0$) that also appear in the clean version x_0 of the dataset:*

$$SAME^{x_0}_{x,y} = SAME(F_{x_0,y}, F_{x_0,y}) = F_{x_0,y} \cap F_{x,y} \tag{4.1}$$

Definition 4.4. Fake FD set. *Given a maximum approximation degree y, $FAKE^{x_0}_{x,y}$ is the set of FDs discovered from a dirty dataset ($x > 0$) that were not valid in the clean dataset x_0:*

$$FAKE^{x_0}_{x,y} = FAKE(F_{x_0,y}, F_{x,y}) = F_{x,y} \setminus \bigcup_{\forall y_0 \leq y} F_{x_0,y_0} \tag{4.2}$$

Definition 4.5. Ghost FD set. *$GHOST^{x_0}_{x,y}$ is the set of FDs discovered from a dirty dataset ($x > 0$) that are valid in the clean dataset x_0 with a certain approximation degree $y_0 \geq 0$, but exist only with a higher approximation degree $y > y_0$ in the dirty dataset:*

$$GHOST^{x_0}_{x,y} = GHOST(F_{x_0,y}, F_{x,y}) = F_{x,y} \cap \bigcup_{\forall y_0 < y} F_{x_0,y_0} \tag{4.3}$$

For generalization, we denote by $F_{x,*}$, the FD set discovered from a dataset with $x\%$ of missing values for all approximation degrees. We denote by $F^{=}_{x,y}$ and $F^{\neq}_{x,y}$, the FD sets discovered with `NULL-EQ` and `NULL-NOT-EQ` semantics respectively. Finally, in presence of the clean dataset ($x_0 = 0$), we define genuine FDs as follows:

Definition 4.6. Genuine FD set. *Given two versions of the same dataset, one containing $x\%$ incomplete values ($x > 0$) and the clean version of the dataset ($x_0 < x$), then $GENUINE_{x,y}^{x_0}$ can be computed as the union of FDs of $SAME_{x,y}^{x_0}$ and $GHOST_{x,y}^{x_0}$.*

Intuitively, genuine FDs discovered from a dirty dataset x are the FDs that hold in the clean version x_0 of the dataset. But generally, we do not have access to the clean dataset. In the next section, we propose a procedure to identify genuine FDs.

4.4 Identifying genuine FDs

As we described previously, the set of FDs that are discovered from an incomplete relation can be *genuine*, *ghost*, or *fake*. Naively using all of the discovered FDs, irrespective of whether they are genuine or not, might be sub-optimal in applications, such as query optimization and data cleaning.

A data analyst would prefer to utilize only the FDs that are either genuine or very likely to be genuine. Our objective is to identify a measure, a *genuineness score*, that can be used to quantify the *"genuineness"* of a given FD . Informally, we would expect for a genuine FD to have a higher genuineness score than non-genuine FDs. Assuming the availability of such a score, we propose the following procedure to identify the set of FDs that are likely to be genuine:

1. Run some exact FD discovery algorithm on the "clean" subset of R that does not have any NULL values; The set of discovered FDs will be a superset of all genuine FDs and can contain both *ghost* and *fake* FDs;

2. Compute the genuineness score for each of the discovered FDs;

3. Prune the list of discovered FDs based on some top-k or a domain-specific threshold whereby all FDs with genuineness score above that threshold are considered *genuine*.

4.5 Probabilistic FD genuineness

In this section, we introduce the probabilistic genuineness score of an FD. Then, we propose an efficient algorithm to exactly compute it. Finally, we suggest a sampling-based approach to efficiently approximate the probabilistic genuineness score.

Probabilistic imputation of missing values

A common strategy for handling missing values is *imputation*. Imputation refers to the statistical process that replaces missing data with substituted values. There has been extensive work in statistics to perform imputation in a robust way [Van Buuren, 2012].

Usually, imputation strategies seek to replace missing data for a given attribute with an estimated value based on the values of other attributes/tuples. For example, a simple

imputation strategy for numeric data is to replace missing data with the median value of all the values of that attribute.

Alternatively, one can use a regression-based approach to estimate the value of an attribute given the values of other attributes. This approach also subsumes various NULL semantics, such as `NULL-EQ` and `NULL-NOT-EQ`. To see why, one can simulate `NULL-EQ` by imputing with NULL values for a given attribute to the same value. Alternatively, one can simulate `NULL-EQ` by imputing all NULL values for a given attribute to a different value.

We now consider a general probabilistic imputation approach to estimate the genuineness score of an FD. For each missing data value, we give a probability distribution over the various values that can be taken. This approach generalizes most of the main imputation strategies and allows us to exploit the connection to the well-studied area of probabilistic databases.

In a probabilistic database, each tuple (or an attribute) is associated with a probability distribution such that it can take different values with different probabilities. A possible world is a specific instantiation of the probabilistic database where each tuple takes a value based on the probability distribution associated with the tuple. As an example, consider a probabilistic database with two tuples t_1 and t_2 that can take two and three values respectively. Then there are totally six possible worlds (by Cartesian product).

In this work, we consider the scenario where the probability distribution is defined over the entire tuple. Note that this approach is more general than the one where the probability distribution is defined over attributes: the former can handle correlated attributes. Table 4.1 shows a probabilistic imputation based on relation R_3 from Figure 4.2, where the probabilities are chosen arbitrarily for expository purposes. For example, the third line of the table can be interpreted as: B and C values of tuple t_3 will be imputed as $t_3[B] = 0$ and $t_3[C] = 1$ with probability 0.2 and $t_3[B] = 1$ and $t_3[C] = 1$ with probability 0.8.

Intuitively, the probabilistic imputation associates with each incomplete tuple a set of possible imputed/complete tuple values it can take with the corresponding probability. This is equivalent to an uncertain tuple in a probabilistic database that is associated with a probability distribution. We also make the tuple independence assumption whereby individual tuples are imputed independently. This is a standard assumption in both probabilistic imputation and probabilistic databases.

Table 4.1: Tuple-level probability distribution for imputation over relation R_3 of the illustrative example.

	A	(**B** , **C**)
t_1	0	(1,1)
t_2	0	$\{(0,0) : 0.12; (0,1) : 0.18; (1,0) : 0.28; (1,1) : 0.42\}$
t_3	1	$\{(0,1) : 0.2; (1,1) : 0.8\}$
t_4	1	(0,1)

The probabilistic genuineness score

Given the setup above, we can now define the probabilistic genuineness score as follows:

Definition 4.7. *The genuineness score of an FD $X \rightarrow A$ is the sum of probabilities of all the possible worlds in which the FD holds.*

Our definition of genuineness generalizes both the *strong* and *weak* FDs [Levene and Loizou, 1998]. A strong FD is one that holds in all possible worlds while a weak FD holds in at least one possible world. Based on our genuineness score definition, we can see that strong FDs have a genuineness of 1 while weak FDs have a genuineness score > 0.

Let us consider how to efficiently compute the probabilistic genuineness score of an FD in a probabilistic imputation setting.

Complete enumeration. The simplest approach to compute the probabilistic genuineness score enumerates all possible worlds, evaluating for each whether the FD holds in that world and then simply summing up the probabilities of all worlds where it does. One can generate the possible worlds in a straightforward manner. The deterministic tuples that have no NULL values exist in all the possible worlds while the tuples with NULL exist with appropriate imputation probability.

Example 4.1. *There are eight possible worlds for the example in Table 4.1 (four for t_2 times two for t_3). Tuples t_1 and t_4 exist in each of them. Tuples t_2 and t_3 take values from the Cartesian product of all possible imputed values. So in possible world w_1, $t_2[B] = 0, t_2[C] = 0$ and $t_3[B] = 0, t_3[C] = 1$. Since this is a tuple-independent probabilistic database, this occurs with probability 0.12×0.2. The last possible world w_8 has $t_2[B] = 1, t_2[C] = 1$ and $t_3[B] = 1, t_3[C] = 1$ with probability 0.42×0.8. One can enumerate other possible worlds and compute its probability in a systematic manner.*

Note that this approach is exact and returns the accurate genuineness score. However, this approach is very expensive as the number of possible worlds grows exponentially in the in the number of tuples in a real-world dataset. We leverage prior work on efficient inference over probabilistic databases [Dalvi and Suciu, 2007; De and Kambhampati, 2010; Koch and Olteanu, 2008] to propose a more efficient algorithm that can compute the *exact* genuineness score by avoiding the enumeration of irrelevant worlds where the FD does not hold.

Efficient enumeration. Consider an FD $X \rightarrow A$ and an arbitrary tuple t_i. Intuitively, we perform two major pruning steps. First, we can notice that when considering the possible worlds where we imputed $t_i[X] = V_X$ and $t_i[A] = V_A$ for some $V_X \in Dom(X), V_A \in Dom(A)$, we no longer need to consider all possible worlds where $t_j[X] = V_X$ and $t_j[A] \neq V_A$ where $j > i$. In other words, the entire set of possible worlds where $t_i[X] = t_j[X] = V_X$, $t_i[A] = V_A$ and $t_j[A] \neq V_A$ will have a contribution of 0 to the genuineness score computation and can be readily pruned. Second, if all the values in a given tuple comply with the FD, then the genuineness score computation does not change whether the tuple is picked or not as its contribution is 1.

Algorithm 3 shows the pseudo-code. Given an FD $X \rightarrow A$, we use the term constraints loosely to denote the set of (X, A) pairs that are valid in the given partial probable world. For example, consider a tuple t_i with $t_i[X] = V_X$ and $t_i[A] = V_A$ where $V_X \in Dom(X), V_A \in Dom(A)$. Then the pair (V_X, V_A) acts as a constraint (denoted by C in Algorithm 3) whereby all possible worlds where another tuple t_j is imputed with same value for X but different value for A is invalid. Please refer to [De and Kambhampati, 2010] for additional details.

Algorithm 3 Find_Probabilistic_Genuineness_Score

Input : D: Imputed database
f: an FD
C: Set of constraints

Output: Genuineness score P of f

13 $P = 0$
14 t = Next tuple to process from D
15 **if** *t does not violate f and C* **then**
16 Estimate_Genuineness_Score($D \setminus t$, f, C)
17 **foreach** *distinct possible (t[LHS], t[RHS]) combination in imputed t* **do**
18 **if** *Possible tuple (t[LHS], t[RHS]) does not violate C* **then**
19 Add constraint (t[LHS], t[RHS]) to C
20 result = Estimate_Genuineness_Score($D \setminus t$, f, C)
21 $P = P$ + Prob(t[LHS], t[RHS]) × result
22 Remove t[LHS], t[RHS] from C
23 **return** P

Example 4.2. *Consider Table 4.1 and try to compute the genuineness score of candidate FD* $A \rightarrow B$. *We can see that tuples* t_1 *and* t_4 *are deterministic and impose the "constraints"* $\{(A = 0, B = 1), (A = 1, B = 0)\}$. *Hence, we need to consider only the set of possible worlds where this set of constraints hold.*

Let us now consider tuple t_2. *Since* $t_2[A] = 0$, $t_2[B]$ *has to be 1 (otherwise it violates the constraints and has a probability of 0). We can see that* $t_2[B] = 1$ *occurs with probability* 0.7 *(*$0.28 + 0.42$*).*

Similarly, $t_3[B]$ *can take only the value of 0 that occurs with probability* 0.2. *The assignment for* t_2 *and* t_3 *happens independently with probability* $0.7 \times 0.2 = 0.14$. *Hence the genuineness score of* $A \rightarrow B$ *is* 0.14.

The efficiency of Algorithm 3 stems from the fact that it avoids enumerating possible worlds where a given FD does not hold. Still, Algorithm 3 is exponential in the cardinality of the domain of the attributes. When the number of attributes involved in the FD is small or when they have low domain cardinality, such as Gender, this approach is orders of magnitude faster than complete enumeration. One can trade the exactness of the probabilistic genuineness score of an FD for an efficient computation by generating a sample of possible worlds.

Monte Carlo sampling of possible worlds. We adapt the Karp-Luby algorithm for approximate model counting that is used for inference over probabilistic databases [Dalvi and Suciu, 2007; Koch and Olteanu, 2008]. In contrast to the complete enumeration approach, we do not enumerate all possible worlds. Instead we generate a sample of possible worlds and compute the genuineness score for each FD from the sample.

Intuitively, we generate different possible worlds in proportion to their likelihood. We then compute the genuineness score as the weighted ratio of the likelihood of all the generated worlds where the FD held to the likelihood of all the generated worlds.

When the size of the sampled possible worlds is large enough, the ratio converges to the correct genuineness score with high probability. Specifically, Dalvi and Suciu [2007]; Koch and Olteanu [2008] showed that if we run the experiment for $N \geq \frac{4n}{\varepsilon^2} \ln \frac{2}{\delta}$, we can guarantee that the probability that the generated estimate being off by more than ε is less than δ. For example, if there are $n = 100$ tuples and we want the genuineness estimate to be within 0.1 of the true value at least 95% of the times, then we need to generate at least 29,778 possible worlds.

Furthermore, one can generate a confidence interval during the execution of the algorithm and can terminate it when the confidence is satisfactory. After sampling N possible worlds with $0 \leq \delta < 1$, we can guarantee that the estimated genuineness score $\tilde{p}$ relates with accurate genuineness score p as follows:

$$P(\tilde{p} \leq (1 - \delta)p) \leq \exp\left(\frac{-N \times p \times \delta^2}{2}\right). \tag{4.4}$$

One can see that the runtime complexity of the algorithm is parameterized by the number of sampled possible worlds N. Since one can evaluate whether a given FD holds in $O(n^2)$, the overall time complexity is $O(N \cdot n^2)$.

In Section 4.7.3, we report on the qualitative performance of sampling-based computation of the probabilistic genuineness score.

4.6 Likelihood-based FD genuineness

An alternate approach to speed up genuineness computation is to limit the expressiveness of the imputation. A number of commonly used imputation and repair strategies are frequency-based (the more frequent a value occurs, the more likely it is to be correct). If one adopts such an imputation strategy, one can design a linear time algorithm to efficiently compute the genuineness score.

Given a candidate FD $X \to A$, we can define its genuineness as the "likelihood" that it is correct. FDs that are more likely would have a higher genuineness score. Note that if the FD $X \to A$ is indeed genuine, we would like its genuineness score (and hence its likelihood) to be 1. However, due to incomplete data, there might be some violating tuples.

Hence, a natural way to define the likelihood of a FD is to compute the fraction of tuples for each distinct value of X where the FD holds. In the following, we present two

approaches adapted from [Wang et al., 2009] to compute efficiently genuineness scores per value and per tuple.

4.6.1 PerValue approach

Given a FD $X \rightarrow A$ and a NULL semantics, we begin by computing the likelihood that the FD holds for each distinct value of X. Consider an arbitrary value $V_X \in Dom(X)$. For all the tuples that have $t[X] = V_X$, we identify the value V_A that occurs the maximum number of times. The likelihood that FD $X \rightarrow A$ holds for the value V_X can be computed as

$$Lik(X \rightarrow A, V_X) = \frac{|V_X, V_A|}{|V_X|} \tag{4.5}$$

where $|V_X, V_A|$ and $|V_X|$ are the number of tuples that have $t[X] = V_X, t[A] = V_A$, and $t[X] = V_X$, respectively.

Note that $Lik(X \rightarrow A, V_X)$ is determined for a specific value V_X. We can compute the likelihood for a FD as the average of the likelihood values for each distinct value V_X. Formally, the genuineness score is computed as

$$Genuineness_{PV}(X \rightarrow A) = \frac{\sum_{V_X \in Distinct(X)} Lik(X \rightarrow A, V_X)}{|Distinct(X)|} \tag{4.6}$$

where $Distinct(X)$ returns all distinct values of X that occur in the relation R.

4.6.2 PerTuple approach

The PerValue approach, while intuitive, has a subtle issue. Consider two groups of tuples for arbitrary values V_X and V_Y. Let $|V_X| = 1,000$ and $|V_Y| = 10$ and assume that $|V_X, V_A| = 800$ and $|V_Y, V_A| = 8$. Using Equation 4.5, the likelihood for both V_X and V_Y are 0.8. Intuitively, we might want to give higher weight to V_X than V_Y. This can be achieved by weighting the likelihood by the frequency of each distinct value V_X. This results in a PerTuple definition of genuineness score computed as

$$Genuineness_{PT}(X \rightarrow A) = \frac{\sum_{V_X \in Distinct(X)} |V_X, V_A|}{\sum_{V_X \in Distinct(X)} |V_X|} \tag{4.7}$$

4.7 Experiments

In this section, we report on our experimental results. First, we expose the phenomenon of fake and ghost FDs when missing values are injected in a clean dataset and we show how missing values can distort the FD discovery result (Section 4.7.2). Second, we apply our algorithms to compute the genuineness score of FDs from various real-world datasets that we have artificially polluted with missing values, and we report the quality

Table 4.2: Clean versions of real-world datasets with the number of (A)FDs discovered (with approximation degree α)

Datasets	[#] Att.	[#] Rows	[#]Distinct (min;max)	[#] Missing	[#]FDs $\alpha = 0$	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$	$\alpha \geq 5$	[#]Total
Iris	5	150	(3;43)	10-40%	5	2	1	1	7	59	80
Abalone	9	4,177	(3;2,429)	10-40%	783	219	122	57	56	1,067	2,313
Computer	9	209	(15;209)	10-40%	3,046	193	199	168	92	1,422	5,129
Glass	10	214	(6;214	10-40%	8,624	1,156	536	166	84	687	11,263
Sensor	8	2,313,681	(137;10,283)	96,733	Skiptuple						
Sensor10			$\rightarrow$ 10		397	29	10	14	11	563	1,024
Sensor100			$\rightarrow$ 100		432	40	10	0	3	539	1,024
Sensor1000			$\rightarrow$ 1000		427	44	7	0	3	543	1,024

performance evaluation of our approach given the true labels of FD discovered from the clean version of the datasets (Section 4.7.3). In particular, we observed:

- Increasing the percentage of missing values in `LHS` attributes of FDs generates fake FDs both for `NULL-NOT-EQ` semantics and `skiptuple` (Section 4.7.2.B);
- Increasing the percentage of missing values in `RHS` attributes of FDs generates fake FDs with `NULL-EQ` semantics or `skiptuple` but ghost FDs for `NULL-NOT-EQ` (Section 4.7.2.C);
- PerValue (**PV**) and PerTuple (**PT**) approximations of the genuineness score have the highest quality performance to discover genuine FDs with `NULL-EQ` semantics. `skiptuple` is not a good strategy to discover genuine FDs (Section 4.7.3.A). We show that our approach is robust and can perform well even under a worst case imputation (Section 4.7.3.B).

Experiments on the real-world Sensor dataset show that our FD-scoring methods can find 100% of genuine FDs that would have been obtained by multiple imputation strategies in very reasonable time, which offers a significant gain over pre- and post-processsing efforts for FD discovery (Section 4.7.4). Regarding runtime performance, the results directly follow the complexity analysis of various algorithms proposed in Sections 4.5 and 4.6.

4.7.1 Experimental setup

In each experiment, we vary (1) the dataset and the characteristics of the discovered FDs in terms of number, set of attributes in `LHS` and `RHS`, and approximation degree; (2) the number and distribution of missing values; and (3) the considered NULL semantics: `NULL-NOT-EQ`, `NULL-EQ`, or `skiptuple`, and (4) the threshold to select the top-k genuine FDs ($k = 10\%, 20\%, 30\%$ and 100% of the total number of FDs discovered).

Datasets

We used five real-world datasets: four are selected from the UCI machine learning repository [Lichman, 2013] and one Sensor dataset from Intel Berkeley Research lab [1]. The first four datasets (used in Section 4.7.2) are originally complete, i.e., without any missing values. The Sensor dataset (used in Section 4.7.4) includes missing values. As shown in Table 4.2, they vary in the number of columns, rows, and discovered FDs, and cover a wide variety of topics, and they are representative in terms of distributional characteristics and distinctness of attribute sets.

We injected a varying percentage of missing values from 5% to 40% in the dataset attributes using one of the three modes: UNIFORM, PARETO, and TARGET. For UNIFORM, we distribute the random injection of missing values uniformly over the set of attributes. For PARETO, we inject randomly 20% of the missing values in 80% of the attributes and 80% in the remaining 20% of the attributes. Using PARETO, we study the impact of a realistically unbalanced distribution of missing values across the attributes and how it can affect FD discovery (e.g., causing more ghost and fake FDs). Using TARGET, we select a subset of attributes involved either in `LHS` or `RHS` for a set of targeted FDs. For each originally complete dataset ($\times 4$), each missing value percentage ($\times 6$), and each distribution mode ($\times 3$), we generate 10 polluted versions to finally obtain $4 \times 6 \times 3 \times 10 = 720$ datasets.

FD discovery

The exact and approximate FDs were discovered from all datasets using the original implementation of the FUN algorithm [Novelli and Cicchetti, 2001]. Once missing values have been injected, we re-ran FD discovery for each NULL semantics and for `Skiptuple`, the case where the tuples containing missing values are skipped in the FD discovery process. We finally analyze $3 \times 720 = 2,160$ FD sets for the experiments of Sections 4.7.2 and 4.7.3 and 18 datasets for Section 4.7.4.

Quality performance evaluation

For the first sets of experiments in Sections 4.7.2 and 4.7.3, we used the ground truth obtained by discovering FDs from the originally clean datasets. We compared the set of FDs discovered before and after injection of missing values. We used the true labels of FDs to compute the traditional measures of precision (P), recall (R), and F1-measure for *Top-k* percent of the discovered FD size defined as:

$$P = \frac{|true\ Genuine\ FDs|}{|Top\text{-}k\ FDs|}$$

$$R = \frac{|true\ Genuine\ FDs|}{|AllFDs|}$$

$$F1_k = \frac{2PR}{(P+R)}$$

[1] `http://db.csail.mit.edu/labdata/labdata.html`

For the case study in Section 4.7.4, since we do not have access to the ground truth, we used the set of FDs discovered from datasets obtained from three most commonly used imputation strategies as a baseline and report the Jaccard coefficient.

Data storage and system setup

We store all discovered FDs with their approximation degrees as well as the attribute sets' distinctness in a MySQL database and perform SQL queries to extract the information we report hereafter. We perform all experiments on a Dell XPS machine with an Intel Core i7-7500U quad-core, 2.8 GHz, 16 GB RAM, Windows 10 64-bit with g++ (GNU).

4.7.2 Ghost and fake FDs phenomenon

A. Impact of NULLs uniformly distributed in LHS and RHS

In this experiment, our goal is to show that ghost and fake FDs exist and have considerable impact on the validity of FD discovery results. First, we randomly injected increasing percentages of missing values uniformly in the attribute list for each clean version of the real-world datasets described in Table 4.2.

We discover the sets of FDs for the full range of approximation degrees in the original, clean version of the dataset as well as from each polluted version in the two NULL semantics and the `Skiptuple` cases.

Figure 4.3 shows two Jaccard coefficients (in Y-axis) averaged over the 10 polluted versions of the Abalone dataset with increasing percentage of missing values (X-axis). The first Jaccard coefficient (`same_approxdeg` as dashed line) is computed as the fraction of the number of common FDs discovered both in the clean dataset and each polluted dataset version for exactly the same approximation degree over the total number of FDs discovered in both datasets. It represents the same FDs as defined in Equation (4.1). The second Jaccard coefficient (`higher_approxdeg` as solid line) represents the fraction of common FDs that have an approximation degree in each dirty version that is higher than in the clean dataset over the total number of FDs discovered in both datasets.

In this figure, we can see that non-exact FDs are the most impacted by the ghost and fake phenomenon. The more missing values are introduced, the more dissimilar sets of FDs with same approximation degree are obtained. Skipping tuples with missing values for FD discovery is clearly not a good option to preserve genuine FDs as the Jaccard coefficients tend to 0 when increasing the percentage of NULL values. We made the same observations of the phenomenon on the other datasets.

B. Impact of NULLs in LHS

To better understand the phenomenon at the attribute set level, we injected missing values with the PARETO mode. We obtained very similar figures to Figure 4.3 in the two cases of NULL semantics. To grasp the phenomenon at a finer grain, we used the TARGET mode over the least and most distinct attributes of the datasets.

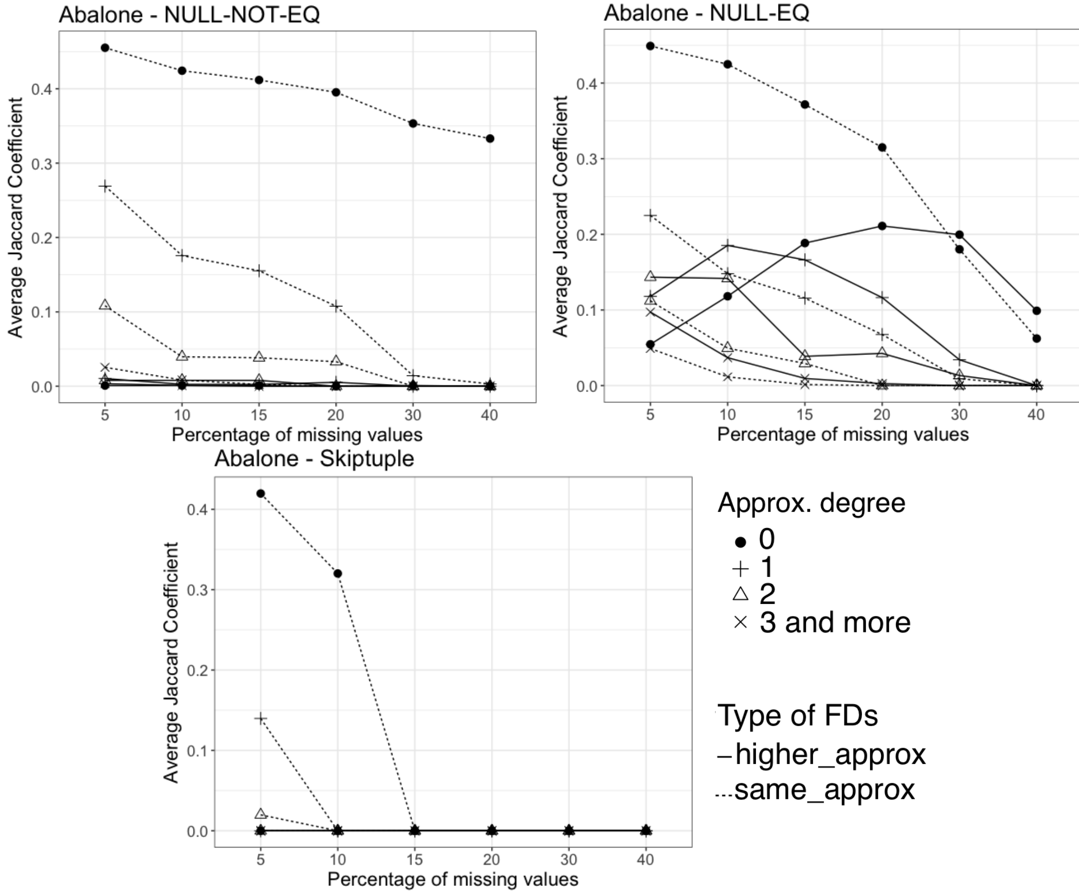


Figure 4.3: Jaccard coefficients with increasing percentage of missing values for Abalone dataset. It measures the similarity between the FD sets discovered from the original, dirty datasets for the same or higher approximation degrees in the 3 cases of NULL semantics: `NULL-NOT-EQ`, `NULL-EQ`, and `skiptuple`.

Figure 4.4 (Top) shows the approximation degree variation (Y-axis) with respect to the percentage of missing values (X-axis) injected in attribute $A10$ (the least distinct attribute) of Glass dataset for each FD having $A10$ in its `LHS` for each NULL semantics cases (similarly Figure 4.4 (Bottom) for `RHS`).

In Figure 4.4 (Top), an increasing percentage of missing values in `LHS` causes a dramatic drop of the approximation degree of all FDs both for `NULL-NOT-EQ` and `Skiptuple` (thus generating fake FDs), whereas for `NULL-EQ` semantics, targeted injection in `LHS` leaves the FDs' approximation degree intact irrespectively of the number of missing values injected.

C. Impact of NULLs in RHS

In Figure 4.4 (Bottom), 150 FDs (not listed due to space limitation) having $A10$ in the RHS are plotted for each NULL semantics. We observe clearly that the increase of their approximation degree is proportional to the increasing of the percentage of missing

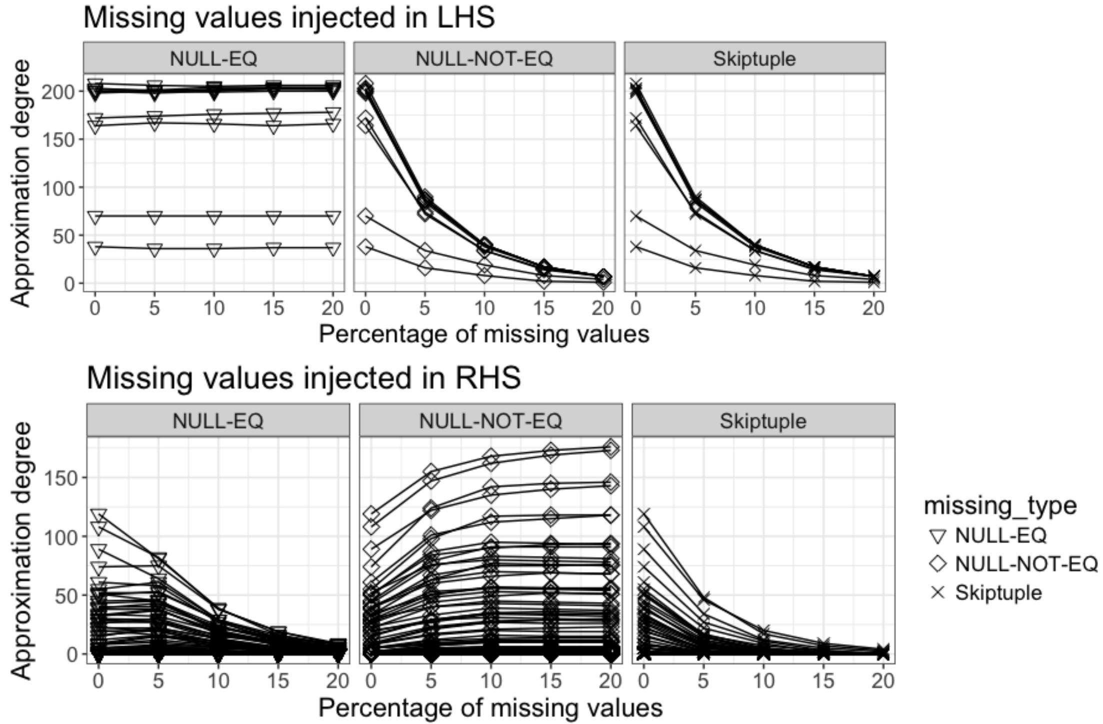


Figure 4.4: Variation of FD approximation degree w.r.t. the NULL semantics when missing values are injected in the least distinct attribute in **LHS** of the FDs and in **RHS** of FDs for the Glass dataset.

values for NULL-NOT-EQ, whereas it decreases significantly in the two other cases with a much steeper slope when the tuples are skipped than for NULL-EQ.

In this case, when more missing value are injected in RHS, depending on the NULL semantics, we can see either the generation of fake FDs (with approximation degrees getting lower for NULL-EQ and Skiptuple) or the disappearance of FDs (becoming ghost with approximation degrees getting higher for NULL-NOT-EQ).

Conclusions

We observe the same phenomenon in all polluted versions of all datasets. This corroborates our conclusions:

- Increasing the distinctness of the LHS attribute set or adding more distinct missing values decreases the approximation degree of the corresponding FDs and more fake FDs appear for NULL-NOT-EQ and Skiptuple;

- Increasing the distinctness of RHS attribute set makes the corresponding FDs become more and more approximate: more genuine FDs disappear and become ghost FDs for NULL-NOT-EQ;

- Decreasing the distinctness of RHS or adding missing values with NULL-NOT-EQ and Skiptuple makes the corresponding FDs become less and less approximate and more fake FDs appear.

4.7.3 Quality evaluation

In this set of experiments, we compute the genuineness scores proposed in Sections 4.5 and 4.6 and report quality performance as accuracy, recall, precision, and F1-measure.

A. PerValue (PV) and PerTuple (PT) genuineness scores

For each polluted version of each dataset, precision, recall, and F1-measure are computed as follows: we select as supposed genuine FDs the ones having PV and PT scores greater than a predefined top-k threshold and we compare them with the true genuine FDs discovered from the clean version of each dataset. This procedure is repeated ten times for averaging the quality metrics.

In Figure 4.5, we report averaged F1-measure of PV and PT scores for top-k genuine FDs discovered from the dirty datasets with $k = 10\%, 20\%$, and 30%. Precision and recall averages are presented in Figure 4.6.

Overall, we observe that `Skiptuple` is the worst performer across all the datasets. All PV and PT scores obtained with the two NULL semantics outperform the scores obtained from `Skiptuple` to a significant extent across all datasets. PV and PT genuineness scores have very close F1-measures except for Iris where PV score reaches 1 despite the increasing percentage of missing values.

We observe that `NULL-EQ` is consistently the best performer, regardless of the missing values percentage. With high percentages of missing values, the difference between scores obtained from `NULL-NOT-EQ` and `NULL-EQ` is greater by more than 10% to 20%.

In conclusion, our PV score combined with `NULL-EQ` semantics can correctly approximate the genuineness of FDs across all datasets. We note that for all datasets, the computation time of PV and PT per FD is negligible (linearly with the dataset size: around 1 second for 100,000 tuples).

B. Sampling-based probabilistic genuineness score.

The qualitative performance of the sampling-based approach is guaranteed to be identical to the PerValue and PerTuple when frequency-based probabilistic imputation is used. Instead, we highlight the robustness of the sampling-based approach by performing a worst-case uniform imputation and show that it still achieves meaningful results.

In uniform imputation, each value in the domain of an attribute is equally likely to be imputed. For example, if the attribute has a domain cardinality of 100, then each of the possible values have 1% probability of being imputed.

We now study how our approach fares under this imputation for the Glass dataset. Applying Eq. (4.4) with $\delta = .95$ and $\varepsilon = .2$ and $\varepsilon = .1$ requires sampling at least 15,931 and 63,724 possible worlds respectively. In Table 4.3, we report the quality metrics of top-10% FDs based on the probabilistic genuineness score (**GS**) computed using 10,000 to 70,000 worlds from exact minimal FDs from `Skiptuple` Glass dataset.

As expected, both precision and recall decreases with increasing missing values. However, our approach has high precision but low recall: we return few FDs but most of the returned FDs are genuine. Given the preponderance of database applications of FDs, returning FDs that are very likely to be genuine is indeed desirable.

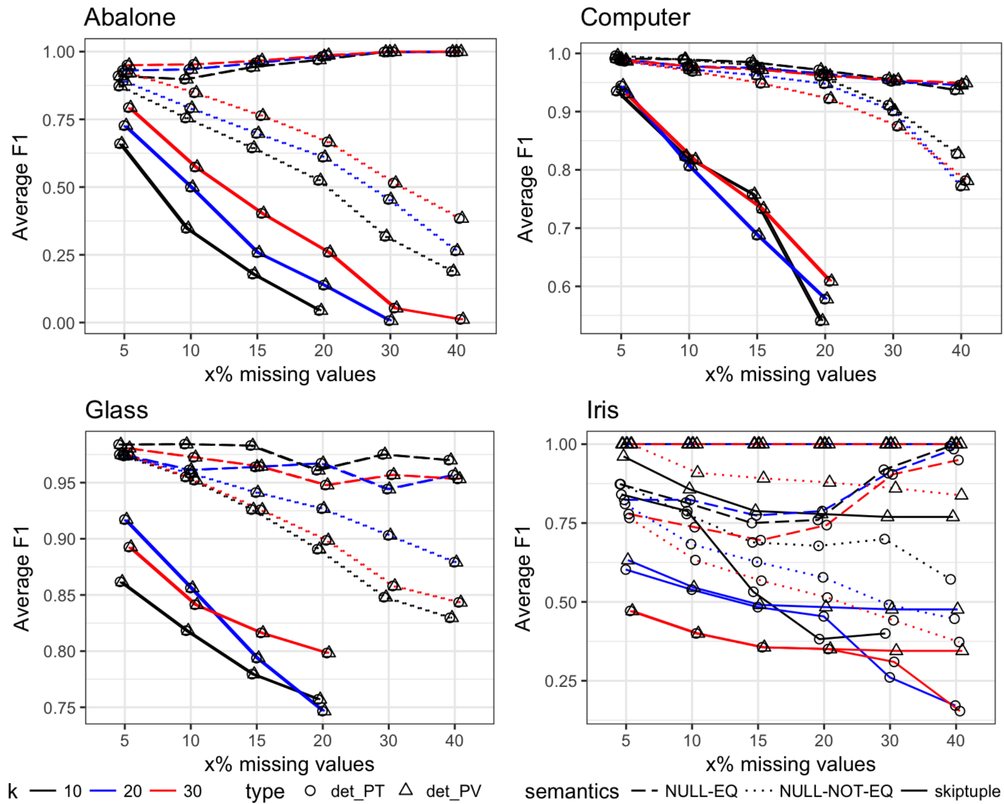


Figure 4.5: Average F1-measures of genuineness scores per value (PV) and per tuple (PT) over 10 runs for `Skiptuple`, `NULL-EQ`, and `NULL-NOT-EQ` semantics and 3 thresholds $k = 10, 20, 30\%$ of the number of FDs discovered from dirty dataset.

Once again, we caution that this result is for the absolute worst case of uniform imputation. If the imputation is reasonably accurate, then the precision/recall will comparable to the PerValue and PerTuple approaches.

4.7.4 Case study on the real-world Sensor dataset

In this set of experiments, we use real-world data collected from 54 sensors deployed including 2,313,681 records identified by a timestamp and five relevant numerical attributes describing the conditions of the monitored rooms such as (`date`, `hour`, `epoch`, `sensorId`, `voltage`, `temperature`, `humidity`, `light`). The dataset includes 96,733 missing values with the distribution given in Table 4.4.

A “1” in the table indicates a non-missing value and a “0” indicates a missing value. There are 2,219,802 observations with non-missing values, and for example, 3 observations with non-missing values except for the variables humidity and light (line 5 of the table). The original number of distinct values per attribute is given in parenthesis. We do not consider the spatio-temporal dimension of the dataset and focus on FD discovery in presence of missing values.

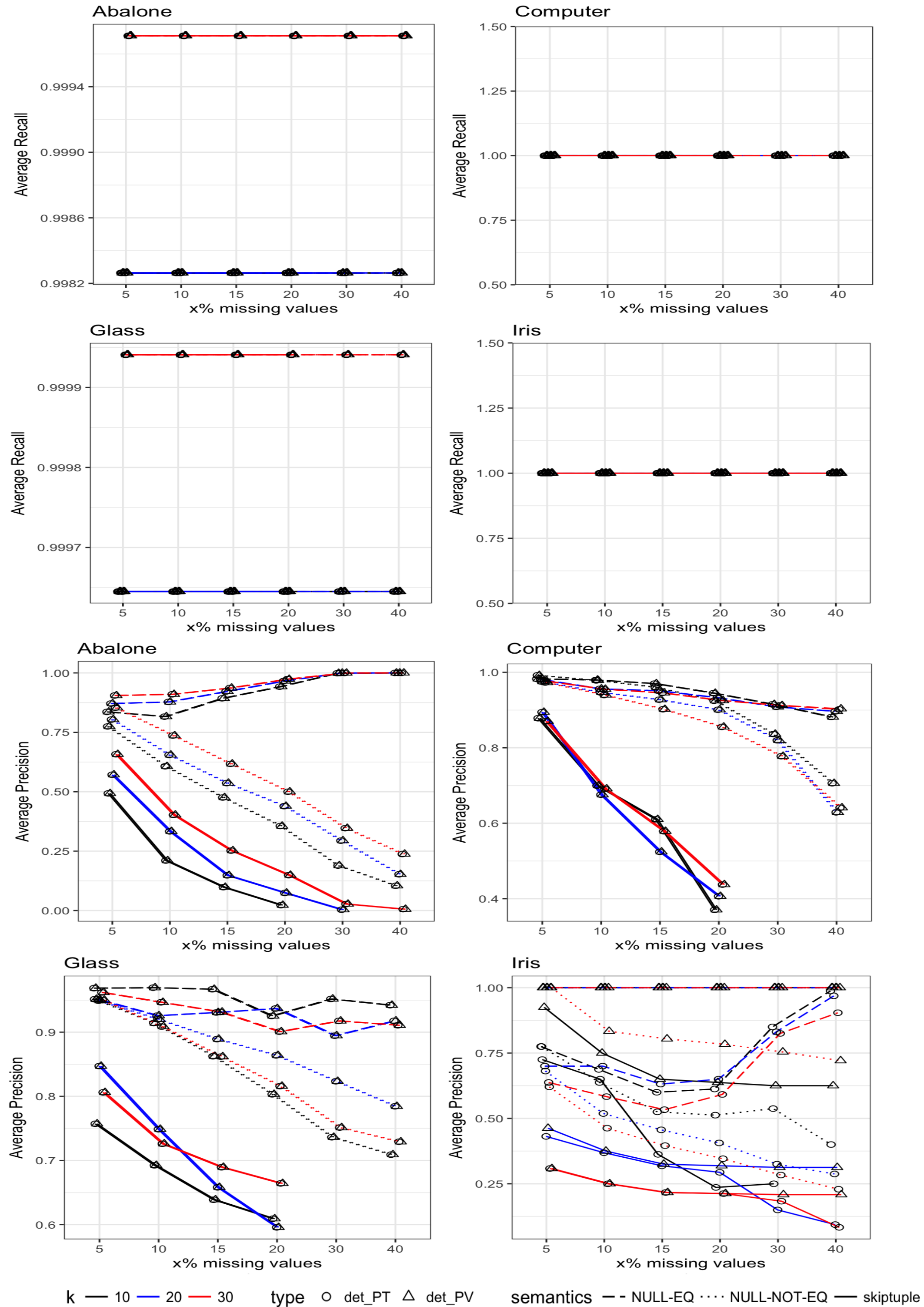


Figure 4.6: Average recall and precision of genuineness scores per value (PV) and per tuple (PT) over 10 runs for `Skiptuple`, `NULL-EQ`, and `NULL-NOT-EQ` semantics and 3 thresholds $k = 10, 20, 30\%$ of the number of FDs discovered from dirty dataset.

Table 4.3: Averaged precision, recall, F1-measure of GS@10% for Glass dataset over 10 runs with 10,000 to 70,000 possible worlds.

Missing (%)	[#] possible worlds	Precision	Recall	F1
5	10,000	0.758	0.082	0.149
	20,000	0.783	0.085	0.154
	50,000	0.778	0.085	0.153
	70,000	0.866	0.094	0.170
10	10,000	0.642	0.078	0.139
	20,000	0.642	0.078	0.139
	50,000	0.657	0.080	0.143
	70,000	0.556	0.068	0.122
20	10,000	0.580	0.033	0.062
	20,000	0.559	0.032	0.060
	50,000	0.570	0.032	0.061
	70,000	0.152	0.009	0.017

Table 4.4: Missing values distribution in Sensor data.

[#]Records	SensorId (61)	voltage (137)	temp. (10,283)	hum. (1,990)	light (143)
2,219,802	1	1	1	1	1
1	1	1	0	1	1
92,975	1	1	1	1	0
1	1	1	0	1	0
3	1	1	1	0	0
373	1	1	0	0	0
526	0	0	0	0	0
2,313,681	526	526	901	902	93,878

To study the effect of attribute cardinality on FD discovery, we transformed the dataset into three binned versions with 10, 100, and 1000 bins, respectively, for the five numerical attributes. We discovered FDs from each binned versions. Table 4.2 (three last lines) reports the numbers of FDs discovered from `Skiptuple` binned version.

We can observe the overlaps of common FDs across various NULL semantics in the Venn diagram for 10 bins in Figure 4.7 (Left). Similar overlaps are observed for 100 and 1000 bins.

Our intuition about the phenomenon of fake and ghost FDs is confirmed as we observed exact FDs, that are present in `Skiptuple`, "disappear" with another NULL semantics, such as the FD `epoch, sensorId,temperature,humidity` $\rightarrow$ `voltage` exact in `Skiptuple` and `NULL-NOT-EQ` versions but with approximation degree 18 in `NULL-EQ`. Next, we computed PV, PT, and GS scores and selected the top-k FDs with $k = 10, 20, 30$, and 100%.

As in many similar application scenarios, we do not have access to the ground truth related to missing values, but a common technique is to apply statistical imputation

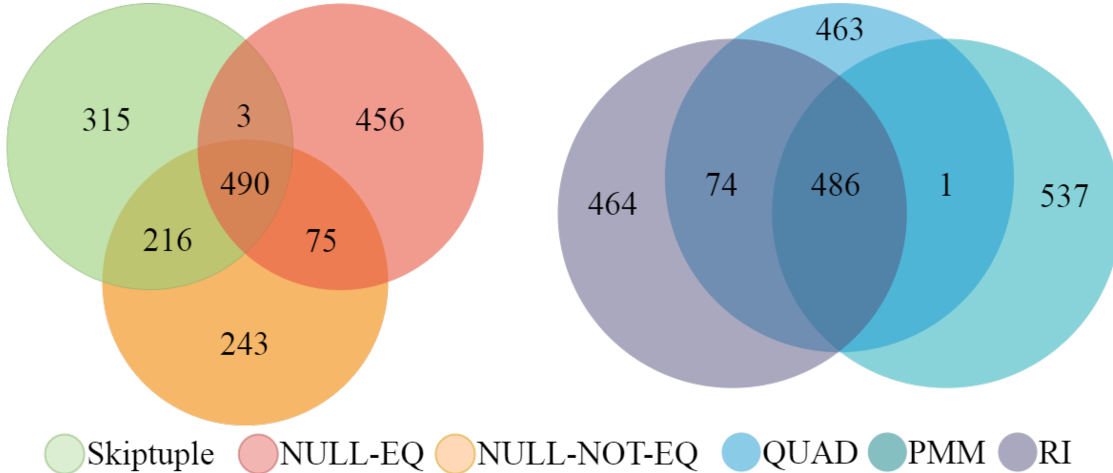


Figure 4.7: Venn diagrams representing the FD sets and their intersections in Sensor 10 Bins dataset between `Skiptuple`, `NULL-EQ` and `NULL-NOT-EQ` versions (Left) and imputation strategies (Right).

methods. Therefore, we applied three imputation strategies to the original Sensor dataset, namely PMM, RI, and QUAD. PMM calculates imputations by predictive mean matching [Van Buuren, 2012]. RI (Random Indicator) estimates an offset between the distribution of the observed and missing data using an algorithm that iterates over the response and imputation models. QUAD is a multivariate imputation technique based on estimating the squared terms [Van Buuren, 2012]. We also applied the same binning strategies to the imputed datasets and discovered three FD sets respectively.

In the absence of ground truth, we can reasonably make the assumption that common FDs across all imputed datasets can be considered as genuine FDs for our comparison purposes. Figure 4.7 (right) represents the overlaps and the set of 486 genuine FDs for imputed Bin 10 Sensor dataset (similar figures for Bin 100 and 1000 are observed). Finally, Figure 4.8 reports the Jaccard coefficient between FDs discovered using our top-k scoring-based methods for various NULL semantics and the set of genuine FDs as $F_{PMM} \cap F_{RI} \cap F_{QUAD}$.

Our results show that with only top-30% of PT- and PV-scoring results obtained from the FD set size of any NULL semantics, around 60% of the set of imputed-genuine FDs can be discovered. PV and PT scores are computed simultaneously for the full Sensor dataset in approximately 21 seconds for 10 Bins, 20 seconds for 100 Bins, and 20 seconds for 1000 Bins for the three cases of NULL semantics and `Skiptuple`. GS score computation times are: 11 min and 35s, 3 hours 32 min, and 3 hours 26 min, respectively.

As a conclusion, the user may choose many different ways to impute missing values and then discover FDs from imputed datasets. However, using our method and in particular PV score regardless of the NULL semantics, the user can obtain the set of

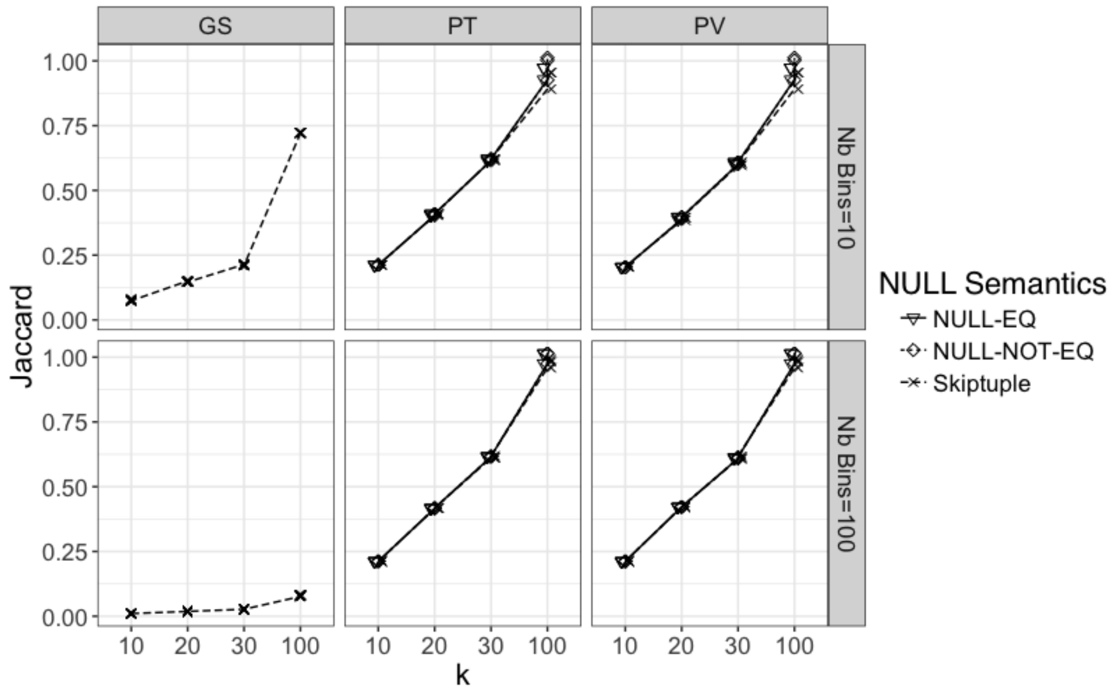


Figure 4.8: Jaccard coefficient between top-k FD sets based on GS, PV, and PT scores and the set of common FDs discovered from imputed datasets using PMM, RI, and QUAD imputation techniques for Sensor dataset with 10 and 100 Bins.

genuine FDs instead of carefully selecting the imputation strategies, spending time for imputation and screening the FDs discovered from multiple imputed datasets.

4.8 Summary

In this chapter, we studied how missing values may impair the final FD discovery results by causing the generation of spurious FDs and the omission of valid FDs at the same time. We formalized the notions of ghost, fake, and genuine functional dependencies.

We proposed a probabilistic approach to quantify the genuineness of FDs and provide an efficient sampling-based computation of genuineness score with accuracy guarantee. We also proposed two algorithms to approximate the genuineness score of FDs based on per value and per tuple granularity levels that could be used by analysts to identify most promising FDs. Experimental results on real-world and semi-synthetic data show high accuracy and efficiency of our scoring model.

For future work, our technique can be extended to the particular case of "disguised" missing values when incorrect default values are misused in replacement of missing values and hardly detectable, which adds complexity into the detection of genuine FDs and anomaly semantics interpretation.

Chapter 5

Conclusion and Outlook

Single-column data profiling provides a basic understanding of relational data by generating a diverse set of descriptive metadata about each column. In this thesis, we focused on the discovery of single column *cardinality* as well as its *header*, representing two important types of single-column metadata. In addition, we analyzed the impact of *the number of* NULL *values* in a column on FDs discovery.

We discussed twelve of the most important algorithms for efficiently estimating the cardinality of a single column, a dataset, or a stream (Chapter 2). We contributed a new classification of these algorithms based on the core method an algorithm uses to estimate the cardinality. We confirmed that some preliminary solutions, such as sampling and hash tables, are valid only when one can scale up the available computational resources. We concluded that none of the twelve estimation algorithms is clearly the best for all datasets and all scenarios. We discussed our insights on which algorithm to use, given the application requirements determined by the three factors: accuracy, memory consumption, and runtime.

In addition to cardinality estimation, we addressed the question of how to find meaningful and coherent headers to a header-less table (Chapter 3). This question is particularly relevant, given the wide range of use cases in which column headers are the main player, such as schema matching, data integration, and KB augmentation. We introduced a fully automated end-to-end schema discovery system, that suggests the combination of the most coherent headers, i.e., the best schema, for a header-less table. Our system conducts an approximate similarity search within a table corpus to suggest headers for each headless-column. It then utilizes statistics of header co-occurrences from a schema corpus to output the best schemata.

This brings us to the last question raised in this thesis: should one trust FDs discovered from incomplete data, given the fact that discovery algorithms are typically defined for correct and complete data. To this end, we showed how missing values may impair the final FD discovery results by causing the generation of spurious FDs and the omission of valid FDs at the same time (Chapter 4). We formalized the notions of ghost, fake, and genuine FDs. We proposed two efficient approaches to approximate the genuineness score of FDs that could be used to identify the most promising FDs.

Apart from continuations, we would also like to point out the limitations of our work and any interesting directions for future work. To this end, we discuss some thoughts on how to extend the solutions proposed throughout this thesis.

Parallel and distributed cardinality estimation. There is ample room for future work to build and evaluate parallel and distributed implementations of cardinality estimation algorithms. Several of the cardinality estimation algorithms discussed in Chapter 2 have characteristics that make them a good candidate for parallelization and distributed environments. They can be divided into three categories: (1) Algorithms whose partial results can be easily merged using a bit-wise OR-operation. (2) Algorithms running several copies of the same algorithm or use several hash functions to improve their accuracy. (3) Algorithms allowing set operations like intersections or unions.

Semantic domain. In Chapter 3, our solution to suggest a header for a column was a value-based solution, i.e., no extra semantic knowledge is needed. But, labeling columns with their semantic domain can be useful in other scenarios that require finding correspondences between columns from different domains, such as table understanding, schema matching, and question answering. Usually, semantic domain detection systems are matching-based: either matching column values with predefined regular expressions or matching column header and values with look-up dictionaries. These systems are not flexible enough to handle dirty data and support only a limited number of semantic domains. Recently, Hulsebos et al. [2019] took a step forward and used modern deep learning methods to detect the semantic domain of a column. Single-column metadata, such as column size, cardinality, and value distribution comprises the main part of the features used for training their machine learning models.

Disguised missing values and genuine FDs. Disguised missing values are data values that are treated as valid values, while it is unknown or non-specified and must be treated as NULL values. For instance, when default values are misused as a replacement of missing values and hardly detectable. Disguised missing values add another level of complexity to the detection of genuine FDs and anomaly semantics interpretation. The straightforward solution to overcome this problem is to detect disguised values and techniques for identifying genuine FDs as proposed in Chapter 4 become applicable. However, the detection of disguised missing values is difficult. A good starting point for continued research in this area is the work by Qahtan et al. [2018].

It is not a secret that single-column profiling results are used as the foundation for multi-column data profiling tasks. In fact, single-column profiling together with multi-column profiling provides a powerful toolkit for data scientists. Although significant advances have been already made in the field of data profiling both by research and industry, future work is still required to address open challenges, such as profiling dynamic data, interactive profiling, profiling results interpretation, better scalability, exploiting modern hardware, and profiling non-relational data. Next, we discuss some of these challenges.

Incremental data profiling. The assumption of static data, i.e., non-changing statistics, limits the effectiveness of many of the current data profiling methods. In other words, any insert, update, or delete operation requires rerunning the whole profiling process. Incremental data profiling algorithms are critical to avoid re-profiling all the data to obtain up-to-date statistics, especially with the increase of volume and velocity of data in modern data management scenarios. Nevertheless, there is already some work in this area. For instance, there are methods to estimate column cardinality even with dynamic data as we discussed in Chapter 2. However, incremental approaches for dependency discovery remain a challenge, as current solutions are too time-consuming to deal with rapidly growing datasets.

User-oriented data profiling. Currently, data profiling methods are not ready to provide an interactive user experience. Some profiling algorithm's runtimes are not suitable when one needs to wait for the results in front of a screen. Furthermore, no interaction with the users is expected during the profiling process, whereas such interaction can be useful in cases, such as data cleansing that requires expert feedback. For example, experts can be consulted to verify a detected error or FD during the profiling process, allowing the algorithm to take this feedback into account as it progresses. In addition, data profiling results can be very large, turning their handling, querying and interpretation into a challenge for the user. Therefore, new profiling algorithms are needed to improve the user experience both during the profiling process and while interpreting their results.

We close this thesis by emphasizing that data profiling remains to be a fundamental data management task considering the multitude of use cases in which data profiling techniques can be beneficial. The importance of data profiling can hardly be understated, particularly with the notable increase in both the volume of data and the number of people working with it.

References

Ziawasch Abedjan, Toni Grütze, Anja Jentzsch, and Felix Naumann. Profiling and mining RDF data with ProLOD++. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1198–1201, 2014.

Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. *Data profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

Divyakant Agrawal, Philip Bernstein, Elisa Bertino, Susan Davidson, Umeshwar Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, H. V. Jagadish, Alexandros Labrinidis, Yannis Sam Madden, Papakonstantinou, Jignesh Patel, M. Raghu Ramakrishnan, Kenneth Ross, Cyrus Shahabi, Dan Suciu, and Shiv Vaithyanathan. Challenges and opportunities with big data: A white paper prepared for the computing community consortium committee of the computing research association, 2012. `http://cra.org/ccc/resources/ccc-led-whitepapers/`, Online; accessed January 16, 2020.

Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.

Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 793–801, 2015.

Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 918–929, 2006.

William Ward Armstrong. Dependency structures of data base relationships. In *Information Processing, Proceedings of the IFIP Congress*, pages 580–583, 1974.

Nikolaus Augsten and Michael Böhlen. *Similarity joins in relational database systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Proceedings of the ACM Web Science Conference*, pages 33–42, 2012.

Antonio Badia and Daniel Lemire. Functional dependencies with null markers. *The Computer Journal*, 58(5):1160–1168, 2015.

Antonio Badia and Daniel Lemire. On desirable semantics of functional dependencies over databases with incomplete information. *arXiv preprint arXiv:1703.08198*, 2017.

Sreeram Balakrishnan, Alon Halevy, Boulos Harb, Hongrae Lee, Jayant Madhavan, Afshin Rostamizadeh, Warren Shen, Kenneth Wilder, Fei Wu, and Cong Yu. Applying WebTables in practice. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2015.

Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proceddings of the International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 1–10, 2002a.

Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002b.

Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *IEEE Computer*, 47(12): 72–80, 2014.

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(5):1217–1230, 2016.

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Active learning of regular expressions for entity extraction. *IEEE transactions on cybernetics*, 48(3): 1067–1080, 2017.

Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 131–140, 2007.

Frank Benford. The law of anomalous numbers. *Proceedings of the American philosophical society*, pages 551–572, 1938.

Philip A Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

Laure Berti-Equille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. Discovery of genuine functional dependencies from relational data with missing values. *PVLDB*, 11(8):880–892, 2018.

Leopoldo E. Bertossi. *Database repairing and consistent query answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

Kevin Beyer, Peter J Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 199–210, 2007.

Kevin Beyer, Rainer Gemulla, Peter J Haas, Berthold Reinwald, and Yannis Sismanis. Distinct-value synopses for multiset operations. *Communications of the ACM*, 52(10): 87–95, 2009.

Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. Methods for exploring and mining tables on wikipedia. In *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics (IDEA@KDD)*, pages 18–26, 2013.

Tobias Bleifuss, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1803–1812, 2016.

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4): 448–461, 1973.

Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

Gerlof Bouma. Normalized (pointwise) mutual information in collocation extraction. *Proceedings of Proceedings of the International Conference of the German Society for Computational Linguistics and Language Technology (GSCL)*, pages 31–40, 2009.

Andrei Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES*, pages 21–29, 1997.

Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 211–222, 2001.

Michael Cafarella, Alon Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. Uncovering the relational web. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2008.

Michael Cafarella, Alon Halevy, Hongrae Lee, Jayant Madhavan, Cong Yu, Daisy Zhe Wang, and Eugene Wu. Ten years of webtables. *PVLDB*, 11(12):2140–2149, 2018.

Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. Relaxed functional dependencies: A survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 268–279, 2000.

Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.

Jiaoyan Chen, Ernesto Jiménez-Ruiz, Ian Horrocks, and Charles Sutton. Colnet: Embedding the semantics of web tables for column type prediction. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 29–36, 2019.

Fei Chiang and Renee J Miller. A unified model for data and constraint repair. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 446–457, 2011.

Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010.

Xu Chu, Ihab F Ilyas, Paolo Papotti, and Yin Ye. Ruleminer: Data quality rules discovery. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1222–1225, 2014.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Graham Cormode. Count-Min sketch. In *Encyclopedia of Database Systems*, pages 511–516. Springer, 2009.

Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the Count-Min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

Graham Cormode, Flip Korn, Shanmugavelayutham Muthukrishnan, and Divesh Srivastava. Space-and time-efficient deterministic algorithms for biased quantiles over data streams. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 263–272, 2006.

Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16(4):523–544, 2007.

Tamraparni Dasu, Theodore Johnson, Shanmugauelayut Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 240–251, 2002.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry*, pages 253–262, 2004.

Herbert Aron David and Haikady Navada Nagaraja. Order statistics. *Encyclopedia of Statistical Sciences*, 2004.

Sushovan De and Subbarao Kambhampati. Defining and mining functional dependencies in probabilistic databases. *arXiv preprint arXiv:1005.4714*, 2010.

Richard De Veaux and David Hand. How to lie with bad data. *Statistical Science*, 20 (3):231–238, 2005.

Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern introduction to probability and statistics: Understanding why and how.* Springer Science & Business Media, 2005.

DemandGen. Assessing the impact of dirty data on sales & marketing performance, 2017. `https://www.zoominfo.com/business/mktg/ebooks/dirtydataebook.pdf`, Online; accessed January 16, 2020.

Hong-Hai Do, Sergey Melnik, and Erhard Rahm. Comparison of schema matching evaluations. In *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Revised Papers*, pages 221–237, 2002.

Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 601–610, 2014.

Igor Douven and Wouter Meijs. Measuring coherence. *Synthese*, 156(3):405–425, 2007.

Ted Dunning and Otmar Ertl. Computing extremely accurate quantiles using t-digests. *arXiv preprint arXiv:1902.04023*, 2019.

Marianne Durand and Philippe Flajolet. LogLog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617, 2003.

Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. Inclusion dependency discovery: An experimental evaluation of thirteen algorithms. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 219–228, 2019.

Bradley Efron. Missing data, imputation, and the bootstrap. *Journal of the American Statistical Association*, 89(426):463–475, 1994.

Vasilis Efthymiou, Oktie Hassanzadeh, Mariano Rodriguez-Muro, and Vassilis Christophides. Matching web tables with knowledge base entities: from entity lookups to entity embeddings. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 260–277, 2017.

David W Embley, Matthew Hurst, Daniel Lopresti, and George Nagy. Table-processing paradigms: a research survey. *International Journal of Document Analysis and Recognition (IJDAR)*, 8(2-3):66–86, 2006.

Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 153–166, 2003.

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3): 281–293, 2000.

Wenfei Fan and Floris Geerts. *Foundations of data quality management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6:1–6:48, 2008.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *VLDB Journal*, 21(2):213–238, 2012.

Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1001–1012, 2018a.

Raul Castro Fernandez, Essam Mansour, Abdulhakim A Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 989–1000, 2018b.

Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. Lazo: A cardinality-based method for coupled estimation of Jaccard similarity and containment. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1190–1201, 2019.

Henning Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

Branden Fitelson. A probabilistic theory of coherence. *Analysis*, 63(3):194–199, 2003.

Philippe Flajolet. On adaptive sampling. *Computing*, 43(4):391–400, 1990.

Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics and Theoretical Computer Science (DMTCS) Proceedings*, AH(1):127–146, 2008.

Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data stream Management: A brave new world*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-540-28608-0.

Gartner. Dirty data is a business problem, not an it problem, 2007. `http://www.gartner.com/newsroom/id/501733`, Online; accessed January 16, 2020.

Phillip Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

Phillip B Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 541–550, 2001.

Phillip B Gibbons. *Data stream management: Processing high-speed data streams*, chapter Distinct-values estimation over data streams. Springer, 2007.

Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

Peter J Haas and Lynne Stokes. Estimating the number of classes in a finite population. *Journal of the American Statistical Association*, 93(444):1475–1487, 1998.

Peter J Haas, Jeffrey F Naughton, S Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 311–322, 1995.

Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *Proceedings of the International Conference on Management of Data (COMAD)*, pages 795–806, 2016.

Felix Halim, Panagiotis Karras, and Roland HC Yap. Fast and effective histogram construction. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1167–1176, 2009.

Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *PVLDB*, 11(4):499–512, 2017.

Anders Haug, Frederik Zachariassen, and Dennis van Liempd. The costs of poor data quality. *Journal of Industrial Engineering and Management*, 4(2):168–193, 2011.

Taher Haveliwala, Aristides Gionis, Dan Klein, and Piotr Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 432–442, 2002.

Stefan Heule, Marc Nunkesser, and Alexander Hall. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 683–692, 2013.

C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

Ming Hua and Jian Pei. Cleaning disguised missing data: a heuristic approach. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 950–958, 2007.

Zhipeng Huang and Yeye He. Auto-detect: Data-driven error detection in tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1377–1392, 2018.

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 392–401, 1998.

Madelon Hulsebos, Kevin Zeng Hu, Michiel A. Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César A. Hidalgo. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 1500–1508, 2019.

Andrew Ilyas, Joana MF da Trindade, Raul Castro Fernandez, and Samuel Madden. Extracting syntactical patterns from databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 41–52, 2018.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 19–30, 2003.

Paul Jaccard. Distribution of alpine flora in the dranses basin and in some neighboring regions. *Bulletin de la Societe vaudoise des Sciences Naturelles*, 37:241–272, 1901.

Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 24–27, 1998.

Theodore Johnson. Data profiling. *Encyclopedia of Database Systems*, pages 604–608, 2009.

Panagiotis Karras and Nikos Mamoulis. Lattice histograms: a resilient synopsis structure. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 247–256, 2008.

Jyrki Kivinen and Heikki Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1): 313–325, 2008.

Henning Köhler, Uwe Leck, Sebastian Link, and Xiaofang Zhou. Possible and certain keys for SQL. *VLDB Journal*, 25(4):571–596, 2016a.

Henning Köhler, Sebastian Link, and Xiaofang Zhou. Discovering meaningful certain keys from incomplete and inconsistent relations. *IEEE Data Engineering Bulletin*, 39 (2):21–37, 2016b.

Sebastian Kruse, Thorsten Papenbrock, Hazar Harmouch, and Felix Naumann. Data anamnesis: Admitting raw data into an organization. *IEEE Data Engineering Bulletin*, 39(2):8–20, 2016.

Abhishek Kumar, Jun Xu, and Jia Wang. Space-code Bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12): 2327–2339, 2006.

Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. The Mannheim search join engine. *Web Semantics: Science, Services and Agents on the World Wide Web*, 35:159–166, 2015.

Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A large public corpus of web tables containing time and context metadata. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 75–76, 2016.

Mark Levene and George Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theoretical Computer Science*, 206(1-2):283–300, 1998.

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 21–30, 2008.

M. Lichman. UCI machine learning repository, 2013. `http://archive.ics.uci.edu/ml`, Online; accessed January 16, 2020.

Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1):1338–1347, 2010.

Hai Liu, Dongqing Xiao, Pankaj Didwania, and Mohamed Y Eltabakh. Exploiting soft and hard correlations in big data query optimization. *PVLDB*, 9(12):1005–1016, 2016.

Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data – a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

David Loshin. *Master data management.* Morgan Kaufmann, 2010.

Qiang Ma, Shanmugavelayutham Muthukrishnan, and Mark Sandler. Frugal streaming for estimating quantiles. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 77–96, 2013.

Michael V Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

Arkady Maydanchik. *Data quality assessment.* Technics publications, 2007.

Mirjana Mazuran, Elisa Quintarelli, Letizia Tanca, and Stefania Ugolini. Semi-automatic support for evolving functional dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 293–304, 2016.

Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 618–629, 2008.

Renée J Miller. Open data integration. *PVLDB*, 11(12):2130–2139, 2018.

David Mimno, Hanna M Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. Optimizing semantic coherence in topic models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 262–272, 2011.

Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. Table union search on open data. *PVLDB*, 11(7):813–825, 2018.

Felix Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2014.

Azade Nazi, Bolin Ding, Vivek Narasayya, and Surajit Chaudhuri. Efficient estimation of inclusion coefficient using HyperLogLog sketches. *PVLDB*, 11(10):1097–1109, 2018.

David Newman, Jey Han Lau, Karl Grieser, and Timothy Baldwin. Automatic evaluation of topic coherence. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 100–108, 2010.

Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 189–203, 2001.

Christopher Palmer, Georgos Siganos, Michalis Faloutsos, , Christos Faloutsos, and Phillip Gibbons. The connectivity and fault tolerance of the internet topology. In *Workshop on Network-Related Data Management (NRDM)*, 2001.

Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28 (2):119–156, 2010.

Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017.

Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with Metanome. *PVLDB*, 8(12):1860–1863, 2015a.

Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB*, 8(10):1082–1093, 2015b.

Eduardo Pena, Eduardo de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3):266–278, 2019.

Rakesh Pimplikar and Sunita Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10):908–919, 2012.

Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Record*, 25(2): 294–305, 1996.

Gil Press. Cleaning big data: most time-consuming, least enjoyable data science task, survey says, 2016. `https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#607a77356f63`, Online; accessed January 16, 2020.

Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. Anmat: Automatic knowledge discovery and error detection through pattern functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1977–1980, 2019.

Abdulhakim A Qahtan, Ahmed Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. Fahes: A robust disguised missing values detector. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 2100–2109, 2018.

Vijayshankar Raman and Joseph M Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 381–390, 2001.

Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

John A Rice. *Mathematical statistics and data analysis.* Cengage Learning, Inc., 2006.

Dominique Ritze, Oliver Lehmberg, and Christian Bizer. Matching HTML tables to DBpedia. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics (WIMS)*, pages 10:1–10:6, 2015.

Dominique Ritze, Oliver Lehmberg, Yaser Oulabi, and Christian Bizer. Profiling the potential of web tables for augmenting cross-domain knowledge bases. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 251–261, 2016.

Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the International International Conference on Web Search and Data Mining (WSDM)*, pages 399–408, 2015.

Yue Shi, Martha Larson, and Alan Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys*, 47(1):3, 2014.

Anshumali Shrivastava. Optimal densification for fast and accurate minwise hashing. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 3154–3163, 2017.

Anshumali Shrivastava and Ping Li. Densifying one permutation hashing via rotation for fast near neighbor search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 557–565, 2014.

Sneha Aman Singh and Srikanta Tirthapura. An evaluation of streaming algorithms for distinct counting over a sliding window. *Frontiers in ICT*, 2:23, 2015.

Shaoxu Song, Aoqian Zhang, Lei Chen, and Jianmin Wang. Enriching data imputation with extensive similarity neighbors. *PVLDB*, 8(11):1286–1297, 2015.

Keith Stevens, Philip Kegelmeyer, David Andrzejewski, and David Buttler. Exploring topic coherence over many models and many topics. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 952–961, 2012.

S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

John W Tukey. *Exploratory data analysis*, volume 2. Reading, Mass., 1977.

S. Van Buuren. *Flexible imputation of missing data.* CRC/Chapman & Hall, 2012.

Daisy Zhe Wang, Xin Luna Dong, Anish Das Sarma, Michael Franklin, and Alon Y. Halevy. Functional dependency generation and applications in pay-as-you-go data integration systems. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2009.

Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

Jingjing Wang, Haixun Wang, Zhongyuan Wang, and Kenny Q Zhu. Understanding tables on the Web. In *Proceedings of the International Conference on Conceptual Modeling (ER)*, pages 141–155, 2012.

Kyu-Young Whang, Brad Vander-Zanden, and Howard Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15:1–15:41, 2011.

Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 97–108, 2012.

Karel Youssefi and Eugene Wong. Query processing in a relational database management system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 409–417, 1979.

Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1):805–814, 2010.

Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. LSH Ensemble: Internet-scale domain search. *PVLDB*, 9(12):1185–1196, 2016.

Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. JOSIE: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 847–864, 2019.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Doktorarbeit mit dem Thema:

Single-column Data Profiling

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Potsdam, den 4. März 2020

Hazar Harmouch