On Discovering and Incrementally Updating Inclusion Dependencies

Nuhad Shaabani

Faculty of Digital Engineering
University of Potsdam, Germany

This dissertation is submitted for the degree of
"Doktor Rerum Naturalium"
(Dr. rer. nat.)
in Computer Science

January 2019

Supervisor:
Prof. Dr. Christoph Meinel (Hasso Plattner Institute, Potsdam, Germany)

Reviewers:
Prof. Dr. Sven Hartmann (TU Clausthal, Germany)
Prof. Dr.-Ing. Ingo Schmitt (BTU Cottbus-Senftenberg, Germany)

Head of the examination board:
Prof. Dr. Andreas Polze (Hasso Plattner Institute, Potsdam, Germany)

Date of Disputation: 09 June 2020

Published online in the
Institutional Repository of the University of Potsdam:
https://doi.org/10.25932/publishup-47186
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-471862

To my son, Taim …

Abstract

In today's world, many applications produce large amounts of data at an enormous rate. Analyzing such datasets for metadata is indispensable for effectively understanding, storing, querying, manipulating, and mining them. Metadata summarizes technical properties of a dataset which rang from basic statistics to complex structures describing data dependencies. One type of dependencies is inclusion dependency (IND), which expresses subset-relationships between attributes of datasets. Therefore, inclusion dependencies are important for many data management applications in terms of data integration, query optimization, schema redesign, or integrity checking. So, the discovery of inclusion dependencies in unknown or legacy datasets is at the core of any data profiling effort.

For exhaustively detecting all INDs in large datasets, we developed S-INDD++, a new algorithm that eliminates the shortcomings of existing IND-detection algorithms and significantly outperforms them. S-INDD++ is based on a novel concept for the attribute clustering for efficiently deriving INDs. Inferring INDs from our attribute clustering eliminates all redundant operations caused by other algorithms. S-INDD++ is also based on a novel partitioning strategy that enables discording a large number of candidates in early phases of the discovering process. Moreover, S-INDD++ does not require to fit a partition into the main memory–this is a highly appreciable property in the face of ever-growing datasets. S-INDD++ reduces up to 50 % of the runtime of the state-of-the-art approach.

None of the approach for discovering INDs is appropriate for the application on dynamic datasets; they can not update the INDs after an update of the dataset without reprocessing it entirely. To this end, we developed the first approach for incrementally updating INDs in frequently changing datasets. We achieved that by reducing the problem of incrementally updating INDs to the incrementally updating the attribute clustering from which all INDs are efficiently derivable. We realized the update of the clusters by designing new operations to be applied to the clusters after every data update. The incremental update of INDs reduces the time of the complete rediscovery by up to 99.999 %.

All existing algorithms for discovering n-ary INDs are based on the principle of candidate generation–they generate candidates and test their validity in the given data instance. The major disadvantage of this technique is the exponentially growing number of database accesses in terms of SQL queries required for validation. We devised MIND_2, the first approach for discovering n-ary INDs without candidate generation. MIND_2 is based on a new mathematical framework developed in this thesis for computing the maximum INDs from which all other n-ary INDs are derivable. The experiments showed that MIND_2 is significantly more scalable and effective than hypergraph-based algorithms.

Zusammenfassung

Viele Anwendungen produzieren mit schnellem Tempo große Datenmengen. Die Profilierung solcher Datenmengen nach ihren Metadaten ist unabdingbar für ihre effektive Verwaltung und ihre Analyse. Metadaten fassen technische Eigenschaften einer Datenmenge zusammen, welche von einfachen Statistiken bis komplexe und Datenabhängigkeiten beschreibende Strukturen umfassen. Eine Form solcher Abhängigkeiten sind Inklusionsabhängigkeiten (INDs), die Teilmengenbeziehungen zwischen Attributen der Datenmengen ausdrücken. Dies macht INDs wichtig für viele Anwendungen wie Datenintegration, Anfragenoptimierung, Schemaentwurf und Integritätsprüfung. Somit ist die Entdeckung von INDs in unbekannten Datenmengen eine zentrale Aufgabe der Datenprofilierung.

Ich entwickelte einen neuen Algorithmus namens S-INDD++ für die IND-Entdeckung in großen Datenmengen. S-INDD++ beseitigt die Defizite existierender Algorithmen für die IND-Entdeckung und somit ist er performanter. S-INDD++ berechnet INDs sehr effizient basierend auf einem neuen Clustering der Attribute. S-INDD++ wendet auch eine neue Partitionierungsmethode an, die das Verwerfen einer großen Anzahl von Kandidaten in früheren Phasen des Entdeckungsprozesses ermöglicht. Außerdem setzt S-INDD++ nicht voraus, dass eine Datenpartition komplett in den Hauptspeicher passen muss. S-INDD++ reduziert die Laufzeit der IND-Entdeckung um bis 50 %.

Keiner der IND-Entdeckungsalgorithmen ist geeignet für die Anwendung auf dynamischen Daten. Zu diesem Zweck entwickelte ich das erste Verfahren für das inkrementelle Update von INDs in häufig geänderten Daten. Ich erreichte dies bei der Reduzierung des Problems des inkrementellen Updates von INDs auf dem inkrementellen Update des Attribute-Clustering, von dem INDs effizient ableitbar sind. Ich realisierte das Update der Cluster beim Entwurf von neuen Operationen, die auf den Clustern nach jedem Update der Daten angewendet werden. Das inkrementelle Update von INDs reduziert die Zeit der statischen IND-Entdeckung um bis 99,999 %.

Alle vorhandenen Algorithmen für die n-ary-IND-Entdeckung basieren auf dem Prinzip der Kandidatengenerierung. Der Hauptnachteil dieser Methode ist die exponentiell wachsende Anzahl der SQL-Anfragen, die für die Validierung der Kandidaten nötig sind. Zu diesem Zweck entwickelte ich MIND_2, den ersten Algorithmus für n-ary-IND-Entdeckung ohne Kandidatengenerierung. MIND_2 basiert auf einem neuen mathematischen Framework für die Berechnung der maximalen INDs, von denen alle anderen n-ary-INDs ableitbar sind. Die Experimente zeigten, dass MIND_2 wesentlich skalierbarer und leistungsfähiger ist als die auf Hypergraphen basierenden Algorithmen.

Contents

1 Introduction and Background **1**

1.1 Data profiling . . . 1
1.2 Data profiling and data mining . . . 2
1.3 Basic notions of the relational data model . . . 3
1.4 Data dependencies . . . 4
1.5 Inclusion dependency . . . 6
1.6 Time complexity of inclusion dependency discovery . . . 9
1.7 Logical implication of inclusion dependencies . . . 11
1.8 Research questions and contributions . . . 11
1.8.1 Discovering n-ary inclusion dependencies . . . 12
1.8.2 Discovering unary inclusion dependencies . . . 14
1.8.3 Incrementally updating inclusion dependencies . . . 15

2 Improving the Efficiency of Inclusion Dependencies Discovery **17**

2.1 Problem statement . . . 17
2.2 Attribute clustering . . . 19
2.3 Attribute clustering and data partitioning . . . 24
2.4 S-INDD++ . . . 29
2.4.1 Overall workflow . . . 29
2.4.2 Computing the partitions . . . 31
2.4.3 Postprocessing of a partition . . . 34
2.4.4 Generating the clusters . . . 35
2.5 Experimental evaluation . . . 39
2.5.1 Setup . . . 39
2.5.2 The effectiveness of the attribute clustering . . . 41
2.5.3 Evaluation of the partitioning strategy . . . 43
2.5.4 Evaluation of the performance . . . 46
2.5.5 Evaluation of the scalability in the number of attributes . . . 49
2.6 Related work . . . 49
2.7 Conclusion and future work . . . 51

3 Incrementally Updating Inclusion Dependencies **53**
3.1 Problem statement 53
3.2 Use cases for incrementally updating INDs 55
3.2.1 Query optimization 55
3.2.2 Schema update and data linkage 56
3.2.3 Data integration 56
3.3 Workflow overview 57
3.4 Attribute clustering operations 59
3.4.1 Merge operator 59
3.4.2 Extract operator 60
3.5 Algorithms 62
3.5.1 Data structures 62
3.5.2 Handling insertion 63
3.5.3 Handling deletion 67
3.5.4 Performance analysis 69
3.5.5 Initializing the data structures 70
3.6 Incrementally updating approximate inclusion dependencies 74
3.7 Scaling out the incremental discovery of INDs 77
3.8 Experimental evaluation 78
3.8.1 Setup 79
3.8.2 Evaluation of the performance 80
3.8.3 Evaluation of cache strategies 82
3.8.4 Comparing with the static discovery 82
3.8.5 Scaling the number of attributes 85
3.8.6 Scaling the number of tuples 85
3.9 Related work 87
3.10 Conclusion and future work 87

4 Discovering Maximum Inclusion Dependencies without Candidate Generation **89**
4.1 Problem statement 89
4.2 Maximum inclusion dependency 91
4.3 Principles of $\textsc{Mind}_2$ 92
4.3.1 Principle 1 92
4.3.2 Principle 2 95
4.3.3 Principle 3 95
4.4 Mind_2 97
4.4.1 Overall workflow 97
4.4.2 Generating unary IND Coordinates 98
4.4.3 Computing maximum INDs between R and S 100

4.4.4 Computing maximum INDs between $\sigma_{ID_R=i}(R)$ and S 103
4.5 Experimental evaluation 103
4.5.1 Setup 103
4.5.2 Evaluation of the performance 104
4.6 Related work 106
4.7 Conclusion and future work 107

5 Conclusion **109**

Bibliography **113**

Chapter 1

Introduction and Background

1.1 Data profiling

In the Big Data era, many applications produce, collect, store, and process ever-growing amounts of data at a fast rate. For instance, Reinsel et al. [2017] have reported that by 2025 the global amount of data will grow to 163 zettabytes (a trillion gigabytes)–it is 10 times the 16.1 zettabytes of data produced in 2016. Such volumes of data come from a variety of sources and in different formats; it can be outdated, undocumented, incomplete, and faulty. Consequently, data scientists initially need to develop a basic understanding of the structure of every received dataset to effectively store, query, and manipulate it.

The activities that systematically extract knowledge of the structure and properties of a dataset are called data profiling, while the gained knowledge is referred to as metadata. Metadata can help to accomplish many data management tasks including data integration, data cleaning, data normalization, query optimization, schema reverse engineering, and many others. These tasks are not only mission-critical, but also very complex [Abedjan et al., 2015; Naumann, 2013].

Metadata can be divided into two categories: data synopses and data dependencies. The first category, data synopses, refers to the traditional profiling tasks like inferring the data types of database attributes, counting the number of their distinct values, estimating the distribution of values in certain attributes, calculation the correlation of attributes, or inferring patterns to describe textual attributes. The second category, data dependencies, concentrates on discovering related attributes as well as bearing open research problems, such as incremental dependency discovery.

This thesis deals with the discovery of inclusion dependencies in unknown datasets–this task is at the core of any data profiling effort [Abedjan et al., 2015; Naumann, 2013]. Inclusion dependency is introduced by Section 1.5, while Section 1.6 shows the time complexity of inclusion dependency discovery. Section 1.7 formulates the logical implication of inclusion dependencies. These sections constitute the basis of Section 1.8 that discusses and formulates the research questions addressed in this thesis.

The other remaining sections are organized as follows: Section 1.2 reflects the discussion on the difference between data profiling and data mining as found in the related work. Section 1.3 reviews the basic notions of the relational data model. An overview of a range of other data dependency types is given in Section 1.4.

1.2 Data profiling and data mining

Data profiling and data mining are closely related approaches to data analysis. A well-defined and accepted difference between data mining and data profiling does not exist [Abedjan et al., 2015].

Rahm and Do [2000], however, distinguish data profiling from data mining as follows:

> *"Data profiling focuses on the instance analysis of individual attributes. It derives information, such as the data type, length, value range, discrete values and their frequency variance, uniqueness, occurrence of null values, typical string patterns (e.g., for phone numbers), etc., providing an exact view of various quality aspects of the attributes. [...] Data mining helps discover specific data patterns in large data sets, e.g., relationships holding between several attributes. This is the focus of so-called descriptive data mining models including clustering, summarization, association discovery and sequence discovery."*

Thus, Rahm and Do [2000] distinguish data profiling from data mining by the number of attributes that are investigated. While this distinction is well-defined, there are, according to Abedjan et al. [2015], several tasks that belong to data profiling, such as inclusion dependencies discovery or functional dependencies discovery, even if they detect relationships between multiple attributes.

Naumann [2013] also characterizes the differences between the two fields as follows:

> *"Data Profiling gathers technical metadata to support data management, while data mining and data analytics discovers non-obvious results to support business management. In this way, data profiling results are information about columns and columns sets, while data mining results are information about rows or rows sets (clustering, summarization, association rules, etc.)."*

Thus, Naumann [2013] formulates a different distinction along two criteria: (i) distinction by the object of analysis (instance versus schema or columns versus rows) and (ii) distinction by the goal of the analysis (description of the data versus new insights beyond the data).

Abedjan et al. [2015], however, have discussed a subset of statistical methods and data mining approaches, such as correlation and association rules, clustering and outliers detections and summaries and sketches, that can be applied to unknown datasets to generate metadata and therefore serves the purpose of data profiling.

1.3 Basic notions of the relational data model

To formulate our statements and definitions in this thesis, we review the basic notions of the relational data model introduced by Codd [1970]. Our review is based on Maier [1983]. Note that even after nearly five decades of its introduction, the relational data model still forms the core of most widely used database management systems [DB-ENGINES, 2019].

Let $\mathbf{A}$ be a non-empty set of attributes. Each attribute $A \in \mathbf{A}$ has a corresponding domain $Dom(A)$, which defines the set of all its possible values. The domains are non-empty sets, finite or countably infinite. A relational schema R is a finite non-empty subset of $\mathbf{A}$. Each attribute $A \in R$ represents a property of the entities described by the schema R. To have a consistent presentation of the relational schema and its data, we assign an order to its attributes. Hence, we refer to the relation schema $R = \{A_1, A_2, \ldots, A_n\}$ as $R[A_1, A_2, \ldots, A_n]$, where $[A_1, A_2, \ldots, A_n]$ is a sequence that defines the order of the attributes of R.

A tuple (or a record) t over R is a mapping from R to $\cup_{A \in R} Dom(A)$ with the restriction that $t(A)$ must be in $Dom(A)$. We refer to $t(A) \in Dom(A)$ as $t[A]$. A relation (or instance) r over a relation schema R is a finite set of tuples over R. Since we consider the attributes of R to be ordered as $[A_1, A_2, \ldots, A_n]$, a relation r over R can also be defined as a finite subset of $Dom(A_1) \times \cdots \times Dom(A_n)$. Notice that ordering the attributes of R adds nothing to the information content of any relation over R.

Let R be a relation schema and let r be a relation over R. For an attribute A in R and for an element a in $Dom(A)$, we refer to the set $\{t \in r \mid t[A] = a\}$ as $\sigma_{A=a}(r)$. That is, $\sigma_{A=a}(r)$ is the selection of the subset of tuples of r with the value a of the specified attribute A. The projection of a tuple $t \in r$ onto $X \subseteq R$ is a tuple t' over X defined as $t'[A] = t[A]$ for each $A \in X$. We denote this projection with $t[X]$. The projection of r onto $X \subseteq R$, written $\pi_X(r)$, is the set $\pi_X(r) = \{t[X] \mid t \in r\}$.

A relational database schema $\mathbf{R}$ over $\mathbf{A}$ is a collection of relation schemas $\{R_1, R_2, \ldots, R_p\}$, where $R_i \subseteq \mathbf{A}$ for each $i \in \{1, \ldots, p\}$ and $\mathbf{A} = \cup_{R_i \in \mathbf{R}}$. A relational database $\mathbf{D}$ over the database schema $\mathbf{R}$ is a collection of relations $\{r_1, r_2, \ldots, r_p\}$ in such a way that for each relation schema R in $\mathbf{R}$ there is a relation r over R in $\mathbf{D}$.

To simplify the formulation in this thesis, we assume without loss of generality that attribute names are unique across all relational schemas. That is,

$$\forall R, S \in \mathbf{R} : R \cap S = \emptyset \tag{1.1}$$

We also define the sets $\mathbf{V}_A$ ($A \in \mathbf{A}$) and $\mathbf{V}$ to ease the notation. For an attribute A, the set $\mathbf{V}_A$ is the values of A that occurs in the relation of the relational schema in which A occurs. That is,

$$\mathbf{V}_A = \{v \in Dom(A) \mid \exists R \in \mathbf{R} : A \in R \wedge v \in \pi_{\{A\}}(r)\} \tag{1.2}$$

The set **V** is then the set of all values of all attributes that occur in **D**. That is,

$$\mathbf{V} = \bigcup_{R_i \in \mathbf{R}} \Big(\bigcup_{A \in R_i} \mathbf{V}_A \Big) \tag{1.3}$$

1.4 Data dependencies

A dependency is a logical statement that describes a relationship among attributes of a database. If the statement is true according to a certain instance of the database, then we say that the dependency is satisfied by the database. That is, a dependency formulates a property of a specific database instance–it is considered as a model for the logical statement defining the dependency [Gyssens, 2009].

Dependencies provide a formal mechanism to express properties expected from (or detected in) the database. If the database is known to satisfy a set of dependencies, this set can be used to (i) improve schema design; (ii) protect the data by preventing certain erroneous updates; and (iii) improve query performance [Abiteboul et al., 1995].

Among the most important dependency types are functional dependencies (FDs) and inclusion dependencies (INDs) [Gyssens, 2009].

Inclusion dependencies (INDs) generalize the notions of referential integrity and foreign keys presented by Codd [1979]. An inclusion dependency states that all tuples of some attribute sequence in one relation are contained in the tuples of some other attribute sequence in the same or in a different relation [Casanova et al., 1984]. Section 1.5 deals with the definition of INDs in detail.

Functional dependencies (FDs) generalize the notions of keys and entity integrity [Codd, 1979]. A relation satisfies a functional dependency $X \rightarrow Y$ (X and Y are subsets of attributes) if the values of a tuple on the set X uniquely determine the values of the tuple on the set Y. The presence of a functional dependency $X \rightarrow Y$ allows a lossless decomposition of the relation into its projections onto $X \cup Y$ and $X \cup \overline{Y}$ to eliminate redundancy–this enables more efficient representation of the relation and avoids update anomalies [Abiteboul et al., 1995]. Approaches for discovering FDs have been developed by Huhtala et al. [1999]; Papenbrock and Naumann [2016]; Yao and Hamilton [2008].

The existence of an FD $X \rightarrow Y$ is a sufficient condition for the lossless decomposition of the relation onto $X \cup Y$ and $X \cup \overline{Y}$, but not a necessary one. That led researchers to introduce the multivalued dependency (MVD): A relation satisfies a MVD $X \twoheadrightarrow Y$ if and only if this relation decomposes losslessly into its projections onto $X \cup Y$ and $X \cup \overline{Y}$ [Maier, 1983]. Discovery algorithms for MVDs have been published by Flach and Savnik [1999]; Savnik and Flach [2000].

A relation, however, can have a lossless decomposition onto three or more subsets of its attributes, but can not have such decomposition onto any two subsets of its attributes. That led to introduce the join dependency: A relation satisfies a join dependency $X_1 \bowtie X_2 \bowtie \ldots \bowtie X_n$ if this relation decomposes losslessly onto $X_1, X_2, \ldots, X_n$ [Maier, 1983].

A generalization of MVDs is full hierarchical dependencies (FHDs): A relation satisfies a FHD precisely if it decomposes losslessly into at least tow of its projections [Hartmann and Link, 2007]. Extensions of MVDs have been found useful for various design problems in advanced data models,

such as the nested relational data model [Fischer et al., 1985] and data models that support nested lists [Hartmann and Link, 2004].

From a data cleaning standpoint, dependencies are used to repair and query inconsistent data. Moreover, they have been revisited and extended to capture more errors in real-world data [Fan, 2008].

In this regard, functional dependencies have been generalized to differential dependencies [Song and Chen, 2011]. A differential dependency has the form $\phi_L[X] \rightarrow \phi_R[Y]$, where $\phi_L[X]$ and $\phi_R Y$ define constraints on distances over values of attributes X and values of attributes Y, respectively. It states that for any two tuples, if their value differences on X, measured by a certain distance metric, satisfy the distance constraints $\phi_L[X]$, then their value differences on Y satisfy also the constraints $\phi_R[Y]$. Specializations of differential dependencies are matching dependencies [Fan et al., 2009] and metric functional dependencies [Koudas et al., 2009]. Both are also an extension of functional dependencies.

Dependencies can be extended by relaxing them. The popular types of relaxed dependencies are partial and conditional dependencies. Caruccio et al. [2016] consider also the value distance constraints defined for differential, matching, and metric functional dependencies as relaxations of functional dependencies.

A partial dependency is a dependency that is satisfied by only a subset of the tuples, e.g., for 98% of the tuples. Partial dependencies are useful for data cleaning: Since they are valid for almost all tuples, they would probably be valid for all tuples if the data was clean. Thus, violating tuples can be identified and cleaned. An algorithm for discovering partial FDs has been suggested by Huhtala et al. [1999]. DeMarchi et al. [2009] have presented a method for discovering partial INDs. The previous two works do not make an explicit difference between the partial dependencies and the approximate discovery of dependencies. As approximate approaches often use sampling or other summarizing techniques, the dependencies discovered by them might be incomplete or might contain false positive dependencies (i.e., dependencies that are not satisfied in the dataset). Examples for approximate discovery of FDs are Ilyas et al. [2004]; Kivinen and Mannila [1995]. For approximate discovery of INDs, Kruse et al. [2017] have presented an approach that finds a superset of valid INDs–it may contain false positives (invalid INDs).

A conditional dependency is a partial dependency extended with conditions that semantically characterize the tuples for which the partial dependency holds. For instance, let $X \rightarrow Y$ be an FD satisfied by a subset r' of a relation r. Then, the conditions in this context are a relation T_p over $X \cup Y$ defined as follows: For each tuple $t_p \in T_p$ and for each $A \in X \cup Y$, $t_p[A]$ is either a value in $Dom(A)$ or a wildcard for some value in $Dom(A)$. The relation T_p is called pattern tableau. The conditional FD $X \rightarrow Y$ with the conditions T_p is then referred to as $(X \rightarrow Y, T_p)$. The relation r satisfies the conditional FD $(X \rightarrow Y, T_p)$ if for each two tuples $t_1, t_2 \in r' \subseteq r$ (r' satisfies the FD $X \rightarrow Y$), and for each pattern $t_p \in T_p$, if $t_1[X]$ and $t_2[X]$ match the pattern $t_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must match the pattern $t_p[Y]$ [Bohannon et al., 2007]. Conditional FDs (CFDs) are introduced by Maher [1997], while conditional inclusion dependencies (CINDs) are proposed by Bravo et al. [2007]. Bauckmann et al. [2012] extended the definition of CINDs to covering and completeness conditions and applied their approach to Wikipedia and DBpedia data. Based on the approach of Bauckmann et al. [2012],

Kruse et al. [2016] developed a distributed system to discover CINDs in RDF data. The discovery of conditional dependencies is harder than the discovery of exact and partial dependencies as identifying the conditions is hard. In this regard, Golab et al. [2008] have shown that the problem of generating the optimal pattern tableau for a given FD is NP-complete.

Order dependencies (ODs) introduced by Ginsburg and Hull [1983] are another class of dependencies. For instance, an order dependency between two attributes A, B of a relation r ($A \neq B$) exists if sorting r on A implies sorting it on B. An algorithm for discovering ODs has been published by Szlichta et al. [2017]. The concept of ODs is extended by Golab et al. [2009] to sequential dependencies (SDs) and to conditional sequential dependencies (CSDs). For a given interval g, a SD $A \rightarrow_g B$ states that if the tuples are sorted on A, the distance between any two consecutive values of B must be within the interval g. For a given SD, Golab et al. [2009] have proposed an algorithm for discovering conditions that identify subsets of A's values for which the given SD is satisfied.

1.5 Inclusion dependency

An inclusion dependency (IND) states that all tuples of some attribute sequence in one relation are contained in the tuples of some other attribute sequence in the same or in a different relation. This means that inclusion dependencies (INDs) generalize the notions of referential integrity and foreign keys. This makes INDs important for many applications, such as data integration [Miller et al., 2001], integrity checking [Casanova et al., 1988], query optimization [Gryz, 1998], or schema redesign [Levene and Vincent, 2000].

The discovery of INDs in unknown datasets is required to identify foreign-primary key relationships, which are a necessity for suggesting join paths, data linkage, and data normalization [Rostin et al., 2009; Zhang et al., 2010].

As INDs express subset-relationships between relations, they are important indicators for redundancies between data sources. The problem of data redundancy arises from those databases that have been independently developed and populated. Consequently, any data integration effort has to deal with such kind of redundancy [Schmitt and Saake, 2005].

With the help of INDs, we can compare data between the relations of the same or different data sources, thereby determining which columns contain overlapping or identical value sets and which are redundant and can be eliminated. Moreover, INDs also help to identify attributes containing the same data but with different names (synonyms) and those having the same name but different semantic (homonyms) [Evoke Software, 2000]. Therefore, the discovery of INDs is an important task for data-integration projects [Bauckmann, 2013; Koeller, 2002; Miller et al., 2001].

The next part of this section introduces the inclusion dependency formally, and then formulates the notations that help us to formulate our statements, definitions, and algorithms.

Definition 1.1. *(Inclusion dependency) Let R and S be two relation schemas, which are not necessarily distinct. For $n \geq 1$, let X be a sequence of n distinct attributes of R and let Y be a sequence of n*

*distinct attributes of S. An **inclusion dependency** (IND) over R and S is an assertion of the form $R[X] \subseteq S[Y]$. Let r and s be two instances of R and S, respectively. An IND $R[X] \subseteq S[Y]$ is **valid** according to r and s if and only if*

$$(\forall t \in r)(\exists t' \in s) : t[X] = t'[Y]$$

The *size* or *arity* of an inclusion dependency $R[X] \subseteq S[Y]$ is defined by $n = |X| = |Y|$. We call an IND with $n = 1$ a *unary* IND (uIND) and an IND with $n > 1$ a *n-ary* IND. An IND $R[X] \subseteq S[Y]$ is trivial if $R = S$ and $X = Y$. When the right-hand side of an IND $R[X] \subseteq S[Y]$ (i.e., Y) confirms a primary key for S, the inclusion dependency is key-based. In this case, the left-hand side (i.e., X) is a foreign key in R.

An inclusion dependency $R[X] \subseteq S[X]$ is merely a statement about two relational schemas that may be valid (true) or invalid (false). A valid IND $R[X] \subseteq S[X]$ describes the fact that there are two instances r of R and s of S such that the projection of r onto X forms a subset of the projection of s onto Y. Note that an IND is defined over two sequences of attributes, not sets, since an IND is not invariant under permutation of the attributes of only one side.

Now, we define some notations that will accompany us throughout the thesis. An inclusion dependency $R[X] \subseteq S[Y]$ is called an IND over a database schema $\mathbf{R}$ if both R and S in $\mathbf{R}$. We refer to the set of all INDs over $\mathbf{R}$ as $\Sigma_{\mathbf{R}}$, or simply as Σ if $\mathbf{R}$ is understood. That is,

$$\Sigma = \{\sigma \mid \exists R, S \in \mathbf{R} : \sigma = R[X] \subseteq S[Y]\} \tag{1.4}$$

For two relational schemas $R, S \in \mathbf{R}$, the set

$$\Sigma_{R \to S} = \{\sigma \in \Sigma \mid \sigma = R[X] \subseteq S[Y]\} \tag{1.5}$$

denotes all INDs $R[X] \subseteq S[Y]$ over R, S. Notice that $\Sigma_{R \to S} \subseteq \Sigma$.

An IND $R[X] \subseteq S[Y]$ over $\mathbf{R}$ is satisfied in a database $\mathbf{D}$ over $\mathbf{R}$ if and only if it is valid according to the instances of R and S in $\mathbf{D}$. We refer to the set of all unary INDs over $\mathbf{R}$ that are satisfied in $\mathbf{D}$ over $\mathbf{R}$ as $\mathbf{I}_{\mathbf{D}}$, or simply as $\mathbf{I}$ if $\mathbf{D}$ is understood. That is,

$$\mathbf{I} = \{\sigma \in \Sigma \mid \sigma \text{ is unary and valid in } \mathbf{D}\} \tag{1.6}$$

According to Equations 1.1 and 1.2, we can reformulate $\mathbf{I}$ as follows:

$$\mathbf{I} = \{A \subseteq B \mid (\exists R \in \mathbf{R})(\exists S \in \mathbf{R}) : A \in R \land B \in S \land \mathbf{V}_A \subseteq \mathbf{V}_B\} \tag{1.7}$$

For two relations $r, s \in \mathbf{D}$ over $R, S \in \mathbf{R}$, the set of all valid uINDs between R and S according to r, s is denoted with $\mathbf{I}_{r \to s}$. That is,

$$\mathbf{I}_{r \to s} = \{A \subseteq B \mid A \in R \land B \in S \land \mathbf{V}_A \subseteq \mathbf{V}_B\} \tag{1.8}$$

Note that $\mathbf{I}_{r\to s} \subseteq \mathbf{I}$. For $R \in \mathbf{R}$ and $A \in R$, the set of all attributes each of whose value set contains the value set of A is denoted with $\mathbf{I}_A$. That is,

$$\mathbf{I}_A = \{B \mid \exists S \in \mathbf{R} : B \in S \wedge \mathbf{V}_A \subseteq \mathbf{V}_B\} \tag{1.9}$$

We refer to the set of all n-ary INDs satisfied in $\mathbf{D}$ over $\mathbf{R}$ as $\Sigma_{\mathbf{D}}$. That is,

$$\Sigma_{\mathbf{D}} = \{\sigma \in \Sigma \mid \sigma \text{ is n-ary and valid in } \mathbf{D}\} \tag{1.10}$$

Notice that $\mathbf{I}_{\mathbf{D}} \cap \Sigma_{\mathbf{D}} = \emptyset$. For two relations $r,s \in \mathbf{D}$ over $R,S \in \mathbf{R}$, the set of all n-ary $R[X] \subseteq S[Y]$ $(n > 1)$ that are valid according to r and s is denoted with $\Sigma_{r\to s}$. That is,

$$\Sigma_{r\to s} = \{\sigma \in \Sigma_{R\to S} \mid \sigma \text{ is n-ary and valid in } \{r,s\}\} \tag{1.11}$$

Notice that $\Sigma_{r\to s} \subseteq \Sigma_{\mathbf{D}}$.

Example 1.1. *To illustrate some sets defined in this section, we consider the database presented in Figure 1.1. We denote the instance of MOVIES with movies and the instance of MY_MOVIES with my_movies.*

The set of all valid uINDs between MY_MOVIES and MOVIES is

$$\mathbf{I}_{my_movies\to movies} = \{Name \subseteq Title, Gerne \subseteq Style\}$$

The set of all valid uINDs between MOVIES and MY_MOVIES is

$$\mathbf{I}_{movies\to my_movies} = \{Style \subseteq Gerne\}$$

The set of all valid uINDs in the entire database is

$$\mathbf{I} = \mathbf{I}_{my_movies\to movies} \cup \mathbf{I}_{movies\to my_movies}$$

The set of all valid n-ary INDs between MY_MOVIES and MOVIES is

$$\begin{aligned}\Sigma_{my_movies\to movies} = \{&MY_MOVIES[Name, Gerne] \subseteq MOVIES[Title, Style]\\ &MY_MOVIES[Gerne, Name] \subseteq MOVIES[Style, Title]\}\end{aligned}$$

Notice that neither $MY_MOVIES[Gerne, Name] \subseteq MOVIES[Title, Style]$ nor $MY_MOVIES[Name, Gerne] \subseteq MOVIES[Style, Title]$ are in $\Sigma_{my_movies\to movies}$ since they are not valid. But both are in $\Sigma_{MY_MOVIES\to MOVIES}$. The set of all valid n-ary INDs between MOVIES and MY_MOVIES is

$$\Sigma_{movies\to my_movies} = \emptyset$$

MOVIES

Title	Style	Director	Year
Dune	Science-Fiction	David Lynch	1984
Titanic	Drama	James Cameron	1997
Titanic	Drama	Jean Negulesco	1953
Dr. Strangelove	Satire	Stanly Kubrick	1963
A.I.	Science-Fiction	Steven Spielberg	2001
Shrek	Animation	Andrew Adamson	2001
2001-A Space Odyssey	Science-Fiction	Stanly Kubrick	1968

MY_MOVIES

Name	Genre
Dune	Science-Fiction
Titanic	Drama
Dr. Strangelove	Satire
A.I.	Science-Fiction
Shrek	Animation

Figure 1.1 Movies database (MDB) adopted from Koeller [2002]

The set of all valid n-ary in the dataset is

$$\Sigma_{MDB} = \Sigma_{my_movies \rightarrow movies} \cup \Sigma_{movies \rightarrow my_movies}$$

1.6 Time complexity of inclusion dependency discovery

Kantola et al. [1992] have shown that it is NP-complete to decide whether a n-ary IND $R[X] \subseteq S[Y]$ is valid, where X contains all attributes of R. To proof that, they defined the following decision problem:

FULL IND EXISTENCE: Let $\mathbf{D} = \{r, s\}$ be a database over the relational schemata $\mathbf{R} = \{R, S\}$, and denote by X the sequence consisting of the attributes of R ($|R| > 1$) in some order. Decide whether there exists a sequence Y consisting of disjoint attributes of S in some order such that the dependency $R[X] \subseteq S[Y]$ is satisfied in $\mathbf{D}$.

After showing that FULL IND EXISTENCE is in NP, they proofed that the NP-complete SUBGRAPH ISOMORPHISM problem is reducible to FULL IND EXISTENCE. SUBGRAPH ISOMORPHISM is defined as follows:

SUBGRAPH ISOMORPHISM: Given graphs $G = (V,E)$ and $H = (V',E')$, decide whether H contains a subgraph isomorphic to G, that is, whether there is an injective mapping $g : V \to V'$ such that for all $v,u \in V$ with $(v,u) \in E$ we have $(g(v),g(u)) \in E'$.

The reduction of the SUBGRAPH ISOMORPHISM problem to the FULL IND EXISTENCE problem is formulated as follows: The relation r corresponding to the graph G has as schema R the set V, and it contains a tuple t_e for each $e = (v,u) \in E$. The tuple t_e is defined by

$$t_e[v] = t_e[u] = 1$$
$$t_e[w] = 0 \text{ for } w \in E \setminus \{u,v\}$$

The relation s corresponding to the graph H is contracted in the same way: The schema S consists of the elements of V', and s has one tuple for each edge in E'.

Kantola et al. [1992] mentioned that H contains a subgraph isomorphic to G if and only if $R[X] \subseteq S[Y]$ is valid in $\mathbf{D} = \{r,s\}$ for some sequence Y of disjoint attributes of S.

In addition to the computational complexity classes of the classical complexity theory [Papadimitriou, 1994], there exist the parameterized complexity classes W[t] ($t \in \mathbb{N}$) that classify computational problems according to their inherent difficulty with respect to multiple parameters of the input or output. The complexity of a parameterized problem is then measured as a function in those parameters. The W hierarchy is defined as $W[t_1] \subseteq W[t_2]$ for all $t_1 < t_2$ [Cygan et al., 2015].

The class FPT (= W[0]) contains the problems that can be solved by algorithms that are exponential only in the size of a fixed parameter while polynomial in the size of the input. Such an algorithm is called a fixed-parameter tractable (FPT), because the problem can be solved efficiently for small values of the fixed parameter. That is, a problem is in FPT if it can be solved in $\mathbf{O}(f(k) \times n^c)$ time, where n is the problem size, k is a fixed parameter, f is a computable function, and c is a constant. For instance, the vertex cover problem can be solved by a FPT-algorithm with the size of the solution as the fixed parameter [Chen et al., 2006].

Bläsius et al. [2016] have proven that the n-ary IND discovery problem is W[3]-complete, which means that under the assumption $W[0] \neq W[3]$ there is no FPT-algorithm for the n-ary IND discovery. Being W[3]-complete makes the n-ary IND discovery one of the hardest natural problems known today. Another W[t]-complete natural problem with $t \geq 3$ is related to supply chain problem [Chen and Zhang, 2006].

Furthermore, Kantola et al. [1992] also stated that the set of all valid uINDs in a given dataset $\mathbf{D}$ over a database relational schema $\mathbf{R}$, i.e., the set $\mathbf{I}$ defined in Equation 1.7, can be computed in time $\mathbf{O}(n^2 m \log m)$, where $n = \sum_{R \in \mathbf{R}} |R|$ and $m = \max\{|r| \mid r \in \mathbf{D}\}$.

Kantola et al. [1992], however, have not presented any algorithm for the discovery of inclusion dependencies.

1.7 Logical implication of inclusion dependencies

Casanova et al. [1984] have defined the problem of inclusion dependency implication as follows:

Definition 1.2. *(Inclusion dependency implication) Let $\Sigma' \subseteq \Sigma$ be a set of INDs over a database schema* **R***, and let $\sigma \in \Sigma$ be a single IND over* **R***. We say that Σ' logically implies σ or that σ is a logical consequence of Σ' if and only if every database* **D** *over* **R** *that satisfies each inclusion dependency in Σ', satisfies σ.*

We then write $\Sigma' \models_{\mathbf{R}} \sigma$, or if **R** is understood, simply $\Sigma' \models \sigma$. That is, $\Sigma' \models \sigma$ if and only if there is no database **D** over **R** such that **D** satisfies every IND in Σ', and such that **D** does not satisfies σ.

To decide $\Sigma' \models \sigma$, Casanova et al. [1984] have introduced the following inference rules:

1. Reflexivity: If X is a sequence of distinct attributes of a relation schema R, then

$$\emptyset \models R[X] \subseteq R[X]$$

2. Projection and permutation: If $[A_1,\ldots,A_m]$ is an attribute sequence of relation schema R_1 and if $[B_1,\ldots,B_m]$ is an attribute sequence of a relation schema R_2, then

$$\{R_1[A_1,\ldots,A_m] \subseteq R_2[B_1,\ldots,B_m]\} \models R_1[A_{i_1},\ldots,A_{i_k}] \subseteq R_2[B_{i_1},\ldots,B_{i_k}]$$

   for each sequence $i_1,\ldots,i_k$ of distinct integers from $\{1,\ldots,m\}$ with $1 \leq k \leq m$.

3. Transitivity: For the relation schemata R_1, R_2, R_3, we have

$$\{R_1[X] \subseteq R_2[Y], R_2[Y] \subseteq R_3[Z]\} \models R_1[X] \subseteq R_3[Z]$$

We say that σ is derivable from Σ' if there is a finite sequence of INDs (over **R**), where (1) each IND in the sequence is either a member of Σ', or else follows from previous INDs in the sequence by an application of the above inference rules, and (2) σ is the last IND in the sequence. We write $\Sigma' \vdash \sigma$ to mean that σ is derivable from Σ'.

Casanova et al. [1984] have shown that $\Sigma' \models \sigma$ and $\Sigma' \vdash \sigma$ are equivalent, which means that if $\Sigma' \vdash \sigma$, then $\Sigma' \models \sigma$, and that if $\Sigma' \models \sigma$, then $\Sigma' \vdash \sigma$. In other words, the previous inference rules are sound and complete. They have also shown that the decision whether $\Sigma' \vdash \sigma$ is decidable and PSPACE-complete.

1.8 Research questions and contributions

For a given dataset **D** over a database schema **R**, the enumeration of all valid INDs in **D** is called the inclusion dependency discovery in **D**, which is divided into two tasks: the unary INDs discovery (i.e., the computation of the set **I** defined in Equation 1.7) and the n-ary INDs discovery (i.e., the computation of the set $\Sigma_{\mathbf{D}}$ defined in Equation 1.10).

1.8.1 Discovering n-ary inclusion dependencies

In principle, computing the set $\Sigma_{\mathbf{D}}$ involves the computation of the set $\Sigma_{r \to s}$ for each two relations $r, s \in \mathbf{D}$ (see Equation 1.11), meaning that

$$\Sigma_{\mathbf{D}} = \bigcup_{r,s \in \mathbf{D}} \Sigma_{r \to s} \tag{1.12}$$

The computation of the set $\Sigma_{r \to s}$, as we have seen in Section 1.6, is NP-complete. Therefore, the core task of discovering n-ary INDs in a given dataset $\mathbf{D}$ is to design algorithms for the discovery of the n-ary INDs between two relations $r, s \in \mathbf{D}$.

To characterize the computation of the set $\Sigma_{r \to s}$, we introduce the partial order $\preceq$ over the set $\Sigma_{R \to S}$ adopted from Mannila and Toivonen [1997] as follows:

Definition 1.3. *(The specialization/generalization relation $\preceq$) For $\alpha, \beta \in \Sigma_{R \to S}$, we say that α is more general than β, denoted by $\alpha \preceq \beta$, if and only if β logically implies α. That is,*

$$\forall \alpha, \beta \in \Sigma_{R \to S} : \alpha \preceq \beta \Leftrightarrow \{\beta\} \models \alpha$$

We also say that β is more specific than α.

Based on the projection and permutation inference rule presented in Section 1.7, the relation $\preceq$ has the following properties:

1. For each two subsequences $W \subseteq X$ and $Z \subseteq Y$ with $|W| = |Z| > 0$, we have

$$R[W] \subseteq S[Z] \preceq R[X] \subseteq S[Y] \tag{1.13}$$

2. If $X = [A_1, \ldots, A_n]$ and $Y = [B_1, \ldots, B_n]$, then for each sequence (permutation) $i_1, \ldots, i_n$ of n distinct integers from $\{1, \ldots, n\}$ we have

$$R[A_{i_1}, \ldots, A_{i_n}] \subseteq S[B_{i_1}, \ldots, B_{i_n}] \preceq R[X] \subseteq S[Y] \tag{1.14}$$

Since the inference rules of inclusion dependencies are sound and complete, we conclude from the first property (i.e., Relation 1.13) that if an IND σ is valid in a given dataset $\mathbf{D}$ over $\mathbf{R}$, then all more general INDs than σ are valid in $\mathbf{D}$. It means that if an IND is invalid in a dataset $\mathbf{D}$, then there is no specialization of it in $\mathbf{D}$. This rule, referred to as the principle of candidate generation, is the basis of all existing algorithms for discovering n-ary INDs: MIND [DeMarchi et al., 2002, 2009], ZIGZAG [DeMarchi and Petit, 2003], and FIND$_2$ [Koeller and Rundensteiner, 2002, 2003]. The central idea is to start from the most general INDs in the corresponding dataset (i.e., the set $\mathbf{I}_{r \to s}$) and then to generate and evaluate more and more special INDs, but not to evaluate those INDs that can not be valid given all valid INDs obtained in earlier iterations.

Algorithm 1: The generate-and-test approach for discovering n-ary INDs. This approach is the basis of MIND, ZIGZAG, and FIND$_2$

Input : r, s, $\mathbf{I}_{\mathbf{r}\rightarrow\mathbf{s}}$
Output: $\Sigma_{r\rightarrow s}$

1 $\Sigma_{r\rightarrow s} \leftarrow \mathbf{I}_{r\rightarrow s}$
2 $i \leftarrow 0$
3 $\Sigma^{i}_{cdd} \leftarrow \emptyset$
4 **repeat**
5 $\quad i \leftarrow i+1$
6 $\quad \Sigma^{i}_{cdd} \leftarrow \{\beta \in \Sigma_{R\rightarrow S} \mid \text{ for all } \alpha \preceq \beta : \alpha \in \Sigma_{r\rightarrow s}\} \setminus \bigcup_{j<i} \Sigma^{j}_{cdd}$
7 $\quad \Sigma_{r\rightarrow s} \leftarrow \{\sigma \in \Sigma^{i}_{cdd} \mid \sigma \text{ is valid in } \mathbf{D}\} \cup \Sigma_{r\rightarrow s}$
8 **until** $\Sigma^{i}_{cdd} = \emptyset$
9 $\Sigma_{r\rightarrow s} \leftarrow \Sigma_{r\rightarrow s} \setminus \mathbf{I}_{\mathbf{r}\rightarrow\mathbf{s}}$

Algorithm 1 formulates the principle of candidate generation for detecting all valid n-ary INDs between two relation r and s in a given dataset **D** over **R** (i.e., computing the set $\Sigma_{r\rightarrow s}$ defined in Equation 1.11). It works iteratively, alternating between candidate generation (Line 6) and evaluation (Line 7) phases. In the generation phase of an iteration i, a set Σ^{i}_{cdd} of new candidate INDs is generated from more general INDs. Then, each candidate is validated against the input relations $\{r,s\}$. The set $\Sigma_{r\rightarrow s}$ will be updated with the those INDs in Σ^{i}_{cdd} that are valid in $\{r,s\}$. In the next iteration i + 1, candidate INDs in Σ^{i+1}_{cdd} are generated using the validated INDs in the previous iterations. The algorithm terminates when no more candidates can be generated. The candidate generation is based on applying only the first property of the relation $\preceq$. The application of the second property is not needed because permutations of both sides do not lead to new INDs. Notice that the computation of the candidate set (Line 6) does not involve the input relations r and s.

The number of the generated candidates determines the number of database queries required to test them. For MIND, this number is always exponential in size of the largest valid IND in the input dataset, while it is exponential for ZIGZAG and FIND$_2$ in the worst case. The reason is that both ZIGZAG and FIND$_2$ differ from MIND in the concrete way of computing the candidate collection Σ^{i}_{cdd} in each iteration (see Sections 4.1 and 4.6 for details).

One of the research questions addressed in this thesis is how to eliminate the exponential number of database accesses required by each of MIND, ZIGZAG, and FIND$_2$ for the candidate validation. We answered this question by developing MIND$_2$ [Shaabani and Meinel, 2016] that computes the set

$$Bd^{+}(\Sigma_{r\rightarrow s}) = \{\alpha \in \Sigma_{r\rightarrow s} \mid \forall \beta \in \Sigma_{R\rightarrow S} : \alpha \preceq \beta \Rightarrow \beta \notin \Sigma_{r\rightarrow s}\} \tag{1.15}$$

without the generation of more general n-ary INDs than any IND in $Bd^{+}(\Sigma_{r\rightarrow s})$. This means that MIND$_2$ computes the set $Bd^{+}(\Sigma_{r\rightarrow s})$ without the generation of any n-ary IND $\sigma \in \Sigma_{r\rightarrow s} \setminus Bd^{+}(\Sigma_{r\rightarrow s})$. Notice that based on the projection and permutations inference rule, knowing the set $Bd^{+}(\Sigma_{r\rightarrow s})$ is

sufficient to derive the complete set $\Sigma_{r \to s}$. The set $Bd^+(\Sigma_{r \to s})$ has been introduced by Mannila and Toivonen [1997], who called it the positive border of the search space.

MIND2 presented in Chapter 4 is the first algorithm for discovering n-ary INDs without candidate generation. Thus, MIND2 eliminates the exponential number of the database accesses required by MIND, ZIGZAG, and FIND2. MIND2 developed new characterizations of the set $Bd^+(\Sigma_{r \to s})$. These characterizations are based on set operations defined on certain metadata that MIND2 generates by accessing the database only 2 $\times$ the number of valid uINDs between r and s (i.e., $2 \times |\mathbf{I}_{r \to s}|$).

1.8.2 Discovering unary inclusion dependencies

In contrast to the discovery of the set $\Sigma_{\mathbf{D}}$ by computing the sets $\Sigma_{r \to s}$ with $r, s \in \mathbf{D}$ separately from each other (see the previous subsection), the discovery of the set $\mathbf{I}$ (i.e., the set of all valid uINDs in $\mathbf{D}$ over $\mathbf{R}$) consists of computing all sets $\mathbf{I}_{r \to s}$ with $r, s \in \mathbf{D}$ at once. This approach is possible because the structure of the set $\mathbf{I}$ (see Equation 1.7) does not involve the relation tuples at all.

Thus, the discovery of all valid uINDs in $\mathbf{D}$ comprises all attribute pairs in $\mathbf{D}$ as uIND candidates at once. This generality enables profiling entirely unknown data sources, even with weak schema definitions. But this generality demands to cope with large numbers of attributes and therefore demands to perform the validation of uIND candidates as efficiently as possible.

In this thesis, we developed S-INDD++ [Shaabani and Meinel, 2018b], an efficient and scalable algorithm for exhaustively discovering all uINDs in large datasets. S-INDD++ presented in Chapter 2 outperforms all existing algorithms for computing the set $\mathbf{I}$ [Bauckmann et al., 2006; DeMarchi et al., 2002, 2009; Papenbrock et al., 2015] by eliminating the shortcomings of them.

The development of S-INDD++ was conducted in two stages. In the first stage, we proposed S-INDD [Shaabani and Meinel, 2015] to improve the scalability of SPIDER proposed by Bauckmann et al. [2006], and to improve the efficiency of the algorithm proposed by DeMarchi et al. [2002, 2009] for detecting uINDs. In this regard, we introduced the concept of the attribute clustering from which the set $\mathbf{I}$ efficiently derivable; we also showed that SPIDER is a special case of S-INDD. The efficiency of inferring the set $\mathbf{I}$ from the attribute clustering is achieved by the elimination of all redundant intersection operations resulting from their inference from the inverted index applied by Bauckmann et al. [2006]; DeMarchi et al. [2002, 2009]; Papenbrock et al. [2015].

In the second stage of S-INDD++'s development [Shaabani and Meinel, 2018b], we designed a new partitioning strategy that helps to discard a large number of attribute candidates in early phases of the discovery process. We also extended the concept of the attribute clustering to decide which attributes has to be discarded based on the attribute clustering of a partition. The effectiveness of S-INDD++'s partitioning strategy manifested itself through the reduction of the runtime of BINDER [Papenbrock et al., 2015] by up to 50 %. Furthermore, in contrast to BINDER, S-INDD++ does not require for processing a partition so that it fits into the main memory–an important property in the face of the ever-growing size of today's datasets.

1.8.3 Incrementally updating inclusion dependencies

A dataset is hardly ever fixed: Analytics-oriented datasets experience periodic updates (typically daily) by frequently appending transactional data to them, and large datasets available on the web are updated in regular time intervals. These data changes cause the metadata to quickly become out-of-date. Accordingly, data profiling methods should be able to efficiently handle such changes, especially without reprocessing the entire dataset [Abedjan et al., 2015; Naumann, 2013; Saha and Srivastava, 2014].

To this end, we developed the first approach for incrementally updating the unary inclusion dependencies in frequently changing datasets [Shaabani and Meinel, 2017, 2018a].

This novel approach, which is presented in Chapter 3, leverages the concept of the attribute clustering by reducing the problem of incrementally updating uINDs to incrementally update of the clusters. The justification of this reduction is the efficiency of inferring uINDs from the attribute clustering introduced by Shaabani and Meinel [2015]. Updating the clusters is achieved by the development of new attribute clustering operations applied after each update of the corresponding dataset. Updating the clusters does not need to access the dataset because of special data structures and caches designed to efficiently support the updating process. We also identified use cases in query optimization, schema update and recode linkage, and data integration that benefit from incremental update of unary inclusion dependencies.

Chapter 2

Improving the Efficiency of Inclusion Dependencies Discovery

2.1 Problem statement

The algorithm proposed by DeMarchi et al. [2002, 2009] for discovering uINDs transforms the dataset into an inverted index, in which every value in the dataset points to the set of all attributes containing that value. Next, the set of all uINDs is computed by intersecting all of those sets. This approach has built the basis for later algorithms for discovering uINDs. However, it is inefficient as an attribute set in the index can be pointed by many different values–this means that the algorithm executes a lot of redundant intersection operations. These operations are costly if the dataset has a large number of attributes sharing a lot of values. Additionally, building the inverted index for the entire dataset at once does not enable the pruning the attributes that are not part of any uIND. Moreover, the inverted index does not fit into the main memory for most real-world datasets.

SPIDER proposed by Bauckmann et al. [2006] has solved the problem of not fitting the inverted index into the main memory. SPIDER writes the values of each attribute to a file after sorting them. Then, it opens all files at once and starts comparing the values in the same way as the merge-sort algorithm. The result of each comparison is a value pointing to the set of all attributes containing that value. SPIDER then uses that set to prune the set of uIND candidates, just like DeMarchi's algorithm. SPIDER, however, has three drawbacks: (i) its scalability in the number of attributes is limited by the underlying operating system because an operating system limits the number of simultaneously open file handles; (ii) it also causes the same number of redundant intersections caused by DeMarchi's algorithm [DeMarchi et al., 2002, 2009]; and (iii) sorting the value sets of attributes of large relations is rather time-consuming and useless for those attributes that are not part of any uIND.

Unlike SPIDER, BINDER presented by Papenbrock et al. [2015] does not require that the value sets of the attributes must be sorted. BINDER divides the dataset into disjoint partitions of equal sizes using a hash-function. Then, by processing the partitions one after the other, BINDER builds the inverted index for each partition and derives the corresponding uINDs. If some attributes are not

part of any uIND detected in a partition, then BINDER discards them from all subsequents partitions. BINDER requires that each partition must fit into the main memory. If a partition does not fit into the main memory, BINDER re-divides it into sub-partitions, thereby causing costly additional I/O-operations. Another shortcoming of BINDER is the generation of uINDs from the inverted index of each partition because of the redundant intersections. Furthermore, analyzing the experimental evaluation of BINDER's partitioning strategy, as reported by Figure 8 in Papenbrock et al. [2015] leads to the following observations: (i) for about 15 % of the datasets, the partitioning strategy can not discard any attribute; (ii) for about 40 % of the datasets, the partitioning strategy can prune some attributes only after processing the first five partitions; and (iii) for most datasets, after discarding some attributes from a partition of a dataset, the partitioning strategy does not discard any further attribute from subsequent partitions of that dataset.

Contributions To eliminate the shortcomings of the uINDs discovery algorithms discussed previously, we developed S-INDD++, a novel algorithm for discovering uINDs in large datasets. In particular, we made the following contributions:

- To eliminate the redundant intersections resulted from generating uINDs from the inverted index, we introduced the attribute clustering concept and showed how to efficiently infer the uINDs from the clusters.
- To compute the attribute clustering, we developed the algorithm S-INDD (short for **S**calable **IN**clusion **D**pendency **D**iscovery). S-INDD solves the scalability problem of SPIDER, which reveals itself as a special case of S-INDD.
- To eliminate the drawbacks of BINDER's partitioning strategy, we developed a configurable partitioning strategy that divides the dataset into disjoint subsets of different sizes to prune a large number of attribute candidates in the early phases of the discovery process.
- Based on our partition strategy, we suggested S-INDD++ that, in contrast to BINDER, does not require fitting any partition into the main memory. Moreover, S-INDD++ reduces the sorting time needed by S-INDD and SPIDER for useless attributes–this means those attributes that can not be part of any non-trivial IND.
- S-INDD++ applies S-INDD to generate the attribute clustering in each partition of the dataset. Therefore, we showed how to compute the attribute clustering of the entire dataset from the clusters of its partitions and how to prune useless attributes based on the clusters of a partition.
- We performed exhaustive experimental evaluations of our approaches by applying them to large datasets with thousands attributes and more than 266 million tuples.
  - We evaluated the attribute clustering concept in terms of the reduction in the intersections needed for computing the uINDs from the inverted index. The reduction gained by inferring the uINDs from the attribute clustering is up to 99.999 %.

- We compared S-INDD's scalability in the number of attributes with that of SPIDER by varying the number of attributes and fixing the number of tuples. For example, by increasing the number of attributes from 6000 to 7000 attributes, SPIDER's runtime increases by38 %, while S-INDD's runtime increases by only 1 %.
- To evaluate S-INDD++ partitioning strategy, we compared it with that of BINDER's strategy in terms of the number of attributes that each of both algorithms prunes after processing each partition of its strategy. The conducted comparisons showed that S-INDD++'s strategy eliminates all shortcomings of BINDER's strategy. For example, S-INDD++'s strategy discardes candidates from each dataset after processing only 1 % of each of them, while BINDER's strategy can not discard any candidate from 80 % of the datasets after processing 30 % of each of them.
- We compared S-INDD++'s runtime with that of BINDER by applying each of both to 10 real-world datasets. In all of these experiments, S-INDD++ significantly outperforms BINDER: S-INDD++ reduces up to 50 % of BINDER's runtime.
- To show how reducing the time needed for completely sorting the value sets of useless attributes improves the discovery runtime, we compared S-INDD++'s runtime with the runtime of S-INDD. In all experiments, S-INDD++ significantly reduces the runtime of S-INDD: S-INDD++ reduces up to 98 % of S-INDD's runtime.

The rest of this chapter is organized as follows: Section 2.2 introduces the attribute clustering concept and shows the computational advantage of inferring the unary INDs from the attribute clustering over generating them from the inverted index. Section 2.3 presents our partitioning strategy and studies the the relationships between the attribute clustering of the entire dataset and the clusters of each of its partitions. Based on the results achieved in Section 2.2 and Section 2.3, Section 2.4 deals with the formulation of both S-INDD++ and S-INDD and the their interaction. Moreover, Section 2.4 shows how S-INDD++ processes each partition and how to divide the dataset based on our partitioning strategy. The results of the conducted experiments are reported by Section 2.5. A detailed overview of the related work is given in Section 2.6. Section 2.7 concludes and discusses future works.

This chapter is based on Shaabani and Meinel [2015, 2018b]. We developed S-INDD in Shaabani and Meinel [2015] and S-INDD++ in Shaabani and Meinel [2018b]. We defined the attribute clustering originally in Shaabani and Meinel [2015] and extended it in Shaabani and Meinel [2017, 2018a].

2.2 Attribute clustering

Definition 2.1. *(Attribute clustering)*[1] *Let* $f : \mathbf{V} \rightarrow 2^{\mathbf{A}}$ *be a function with the property:*

$$(\forall v \in \mathbf{V})(\neg\exists A \in \mathbf{A}) : (v \in \mathbf{V}_A \wedge A \notin f(v)) \text{ or } (v \notin \mathbf{V}_A \wedge A \in f(v)) \tag{2.1}$$

[1] See Sections 1.3 and 1.5 in Chapter 1 to recall some basic notations

Table 2.1 Running example

A	B	C	D
a	a	f	a
b	b	e	a
b	c	f	c
d	d	f	c

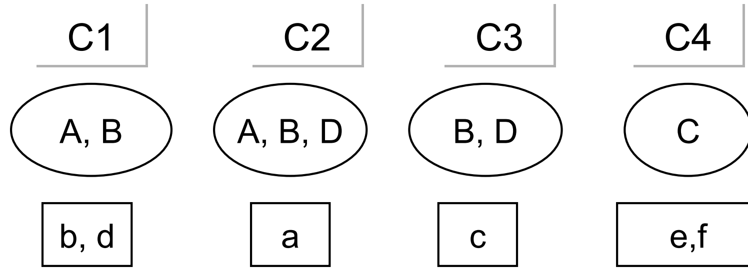


Figure 2.1 Attribute clustering based on the data of Table 2.1

That is, $f(v)$ is the maximum set of attributes $A \in \mathbf{A}$ with $v \in \mathbf{V}_A$. We call the image of f:

$$\mathbf{AC} = f(\mathbf{V}) = \{\mathbf{C} \subseteq \mathbf{A} \mid \exists v \in \mathbf{V} : f(v) = \mathbf{C}\} \tag{2.2}$$

the attribute clustering over the set $\mathbf{V}$*. We call the function f the generator of* $\mathbf{AC}$.

The previous definition implies that each attribute $A \in \mathbf{A}$ must be contained in at least one cluster and that a very value $v \in \mathbf{V}$ must belong to only one cluster.

Example 2.1. *The attribute clustering of the dataset presented in Table 2.1 is*

$$\mathbf{AC} = \{\{A,B\},\{A,B,D\},\{B,D\},\{C\}\}$$

Figure 2.1 illustrates its generator. For example, we have

$$f(b) = f(d) = \{A,B\} = \mathbf{C}_1$$

Thus, attributes A and B shape the cluster $\mathbf{C}_1$ since both share the values $\{b,d\}$ that can not be shared by any superset of $\mathbf{C}_1$.

The following lemma shows the existence of the attribute clustering for any dataset, while Lemma 2.2 states that there can be only one attribute clustering for a dataset.

Lemma 2.1. *For a database instance* $\mathbf{D}$ *over a database schema* $\mathbf{R}$*, there is always an attribute clustering in terms of Definition 2.1.*

Proof. We have to find a function $f : \mathbf{V} \to 2^{\mathbf{A}}$ that satisfies Property 2.1. We define f as follows:

$$f(v) = \bigcup_{A \in \mathbf{A} \wedge v \in \mathbf{V}_A} \{A\} \tag{2.3}$$

Since for every value $v \in \mathbf{V}$ there is some attribute $A \in \mathbf{A}$ whose value set $\mathbf{V}_A$ contains v (see Equations 1.2 and 1.3), $f(v)$ exists for every $v \in \mathbf{V}$.

Furthermore, the set $f(v)$ in Equation 2.3 is the union of all attributes A with $v \in \mathbf{V}_A$, which implies that for any attribute $A \in \mathbf{A}$, $A \in f(v)$ if $v \in \mathbf{V}_A$ and $A \notin f(v)$ if $v \notin \mathbf{V}_A$. Thus, f satisfies Property 2.1. □

Lemma 2.2. *A database instance* $\mathbf{D}$ *over a database schema* $\mathbf{R}$ *can have only one attribute clustering over the value set* $\mathbf{V}$.

Proof. Based on Lemma 2.1, there is an attribute clustering $\mathbf{AC}$ over $\mathbf{V}$. We assume that there a different attribute clustering $\mathbf{AC}'$ over $\mathbf{V}$. Let f be the generator of $\mathbf{AC}$ and let f' be the generator of $\mathbf{AC}'$. Based on this assumption, there is $v \in \mathbf{V}$ for which $f(v) \neq f'(v)$.

That means that there is $A \in \mathbf{A}$ with $v \in \mathbf{V}_A$ and for which either $(A \in f(v) \wedge A \notin f'(v))$ is valid or $(A \notin f(v) \wedge A \in f'(v))$ is valid.

The first case means that f' can not be the generator of $\mathbf{AC}'$. The second case means that f can not be the generator of $\mathbf{AC}$. Thus, our assumption is wrong. □

The next lemma, which is a generalization of Property 1 formulated in DeMarchi et al. [2002], states that for each two different attributes A, B, the set of A's values is included in the set of B's values if and only if the intersection of all clusters containing A contains B. In other words, we have the following inference rule: for any attribute A, the set of all attributes including A is the intersection of all clusters containing A.

Lemma 2.3. *Let* $\mathbf{AC}$ *be the attribute clustering over* $\mathbf{V}$. *The following rule holds.*

$$\forall A, B \in \mathbf{A} : \mathbf{V}_A \subseteq \mathbf{V}_B \Leftrightarrow B \in \bigcap_{\mathbf{C} \in \mathbf{AC}, A \in \mathbf{C}} \mathbf{C}$$

Proof. 1) "⇒": Let $\mathbf{C} \in \mathbf{AC}$ be a cluster with $A \in \mathbf{C}$. Based on Definition 2.1, there is some $v \in \mathbf{V}_A$ with $f(v) = \mathbf{C}$. According to Property 2.1, B must be in $f(v)$ since $\mathbf{V}_A \subseteq \mathbf{V}_B$ (i.e., $v \in \mathbf{V}_B$). Thus, each cluster that contains A must contain B if the value set of A is included in the value set of B.

2) "⇐": We assume $\mathbf{V}_A \not\subseteq \mathbf{V}_B$. That means that there is some $v \in \mathbf{V}$ with $v \in \mathbf{V}_A$ and $v \notin \mathbf{V}_B$, meaning that $f(v)$ contains A and can not contain B according to the definition of f. Therefore, B can not be in $\cap_{\mathbf{C} \in \mathbf{AC} \wedge A \in \mathbf{C}} C$. Thus, our assumption is wrong. □

Example 2.2. *Consider Figure 2.1. We have* $\mathbf{V}_D \subseteq \mathbf{V}_B$ *since*

$$B \in \bigcap_{D \in \mathbf{C}} \mathbf{C} = \mathbf{C}_2 \cap \mathbf{C}_3 = \{A, B, D\} \cap \{B, D\} = \{B, D\}$$

While $\mathbf{V}_B \not\subseteq \mathbf{V}_D$ *since*

$$D \notin \bigcap_{B \in \mathbf{C}} \mathbf{C} = \mathbf{C}_1 \cap \mathbf{C}_2 \cap \mathbf{C}_3 = \{A,B\} \cap \{A,B,D\} \cap \{B,D\} = \{B\}$$

The next two lemmas show the computational advantage of deriving the unary INDs from the clusters over deriving them from the inverted index applied by Bauckmann et al. [2006]; DeMarchi et al. [2002, 2009]; Papenbrock et al. [2015].

Lemma 2.4. *In the worst case, the number of intersections required to compute the set of all valid uINDs (i.e., the set* $\mathbf{I}$ *defined by Equation 1.7 in Chapter 1) from the clusters is*

$$(|\mathbf{AC}| - 1) \times |\mathbf{A}|$$

Proof. For an attribute $A \in \mathbf{A}$, let $\mathbf{I}_A$ be the set of all attributes whose value sets contain the value set of A. Then, based on Lemma 2.3 we have

$$\mathbf{I}_A = \{B \in \mathbf{A} \mid \mathbf{V}_A \subseteq \mathbf{V}_B\} = \bigcap_{\mathbf{C} \in \mathbf{AC} \wedge A \in \mathbf{C}} \mathbf{C}$$

The worst case occurs when each cluster $\mathbf{C} \in \mathbf{AC}$ contains A. In this case, the number of intersections required to calculate $\mathbf{I}_A$ is

$$|\mathbf{AC}| - 1$$

The set $\mathbf{I}$ can be expressed as

$$\mathbf{I} = \{A \subseteq B \mid A, B \in \mathbf{A} \wedge B \in \mathbf{I}_A\}$$

Thus, in the worst case, the number of intersections needed to derive $\mathbf{I}$ from the clusters is

$$(|\mathbf{AC}| - 1) \times |\mathbf{A}|$$

□

Notice that in the case in which $\mathbf{AC}$ contains only one cluster $\mathbf{C}$, computing the set $\mathbf{I}$ does not require any intersection operation because, in this special case, $\mathbf{I}$ equals the set $\{A \subseteq B \mid A, B \in \mathbf{C}\}$.

Lemma 2.5. *Computing* $\mathbf{I}$ *(i.e., the set of all valid uINDs) from the inverted index results in up to*

$$(|\mathbf{V}| - |\mathbf{AC}|) \times |\mathbf{A}|$$

redundant intersections.

Proof. For each cluster $\mathbf{C}_i$ ($1 \leq i \leq |\mathbf{AC}|$), we define the set

$$\mathbf{Q}_i = \{v \in \mathbf{V} \mid f(v) = \mathbf{C}_i\}$$

Since each value $v \in \mathbf{V}$ must belong to a cluster and can not have two different clusters, the sets $\mathbf{Q}_i$ $(1 \le i \le |\mathbf{AC}|)$ have the following two properties:

$$\mathbf{V} = \bigcup_{1 \le i \le |\mathbf{AC}|} \mathbf{Q}_i \tag{2.4}$$

$$\mathbf{Q}_i \cap \mathbf{Q}_j = \emptyset \text{ for } i \neq j (1 \le i, j \le |\mathbf{AC}|) \tag{2.5}$$

From 2.4 and 2.5, we conclude that $\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_{|\mathbf{AC}|}$ divide the set $\mathbf{V}$ into $|\mathbf{AC}|$ disjoint partitions. Therefore, we can formulate the inverted index $\mathbb{B}$ defined by DeMarchi et al. [2009] as

$$\mathbb{B} = \bigcup_{1 \le i \le |\mathbf{AC}|} (\mathbf{Q}_i \times \{\mathbf{C}_i\})$$

Let $\mathbf{I}_A$ be the set of all attributes whose value sets contain the value set of A. Computing $\mathbf{I}_A$ based on Proposition 1 formulated in DeMarchi et al. [2009] is equal to computing $\mathbf{I}_A$ from the formula

$$\mathbf{I}_A = \bigcap_{1 \le i \le |\mathbf{AC}|} \Big(\bigcap_{A \in \mathbf{C}_i \wedge (v, \mathbf{C}_i) \in \mathbb{B}_i} \mathbf{C}_i \Big) \tag{2.6}$$

where

$$\mathbb{B}_i = \mathbf{Q}_i \times \mathbf{C}_i \subseteq \mathbb{B}$$

However, the intersections in each term

$$\bigcap_{A \in \mathbf{C}_i \wedge (v, \mathbf{C}_i) \in \mathbb{B}_i} \mathbf{C}_i$$

of Formula 2.6 are redundant because the result of them is known –namely, the set $\mathbf{C}_i$ itself. The number of these intersections is $|\mathbf{Q}_i| - 1$. Since A might be contained in each cluster $\mathbf{C}_i$ $(1 \le i \le |\mathbf{AC}|)$, the total number of redundant intersections might reach

$$\sum_{1 \le i \le |\mathbf{AC}|} (|\mathbf{Q}_i| - 1) = \sum_{1 \le i \le |\mathbf{AC}|} |\mathbf{Q}_i| - |\mathbf{AC}| = |\mathbf{V}| - |\mathbf{AC}|$$

Therefore, there are up to

$$(|\mathbf{V}| - |\mathbf{AC}|) \times |\mathbf{A}|$$

redundant intersections if we compute $\mathbf{I}$ from the inverted index, because computing $\mathbf{I}$ requires the calculation of all $\mathbf{I}_A$ $(A \in \mathbf{A})$. □

Notice that the computation of $\mathbf{I}$ from the attribute clustering depends on the number of clusters, while computing it from the inverted index depends on the number of distinct values in the dataset. Moreover, since the cost of an intersection operation in both approaches is $\mathbf{O}(|\mathbf{A}|)$, the cost of deriving

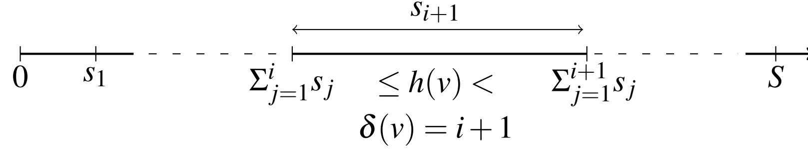


Figure 2.2 Illustration of the partitioning function δ

the set **I** from the clusters is $\mathbf{O}(|\mathbf{AC}| \times |\mathbf{A}|^2)$, while the cost of deriving it from the inverted index is $\mathbf{O}(|\mathbf{V}| \times |\mathbf{A}|^2)$.

2.3 Attribute clustering and data partitioning

In this section, our goal is to develop a technique that makes it possible to identify a large number of useless attributes by only processing a few small subsets of the dataset. In this regard, the useless attributes are those attributes that are not part of any valid IND in the dataset.

Dividing the dataset by a straightforward application of a hash-function results in subsets in equal sizes, which means two issues: (i) having all subsets in equal and small sizes increases the number of subsets, which degrades the performance because of the increase in I/O operations and (ii) having all subsets in equal and large sizes is the opposite of our goal of discarding useless attributes as early as possible by firstly processing a few and small parts of the data. These two design issues lead us to introduce the partitioning function δ in Definition 2.3. Furthermore, having the dataset divided into disjoint partitions also leads us to study (i) how to compute the attribute clustering of the entire dataset from the clusters of its partitions (Lemma 2.6) and (ii) how to identify some useless attributes in the entire dataset based on the attribute clustering of a partition (Lemma 2.7).

Definition 2.2. *(Data partitioning) For a given $n \geq 1$, a partitioning of the dataset $\mathbf{D}$ is a collection $\mathbf{P} = \{\mathbf{P}_i \mid 1 \leq i \leq n\}$ with the following properties:*

1. $\mathbf{P}_i = \{\mathbf{V}_{i,A} \mid A \in \mathbf{A} \wedge \mathbf{V}_{i,A} \subseteq \mathbf{V}_A\}$*, for each* $i \in \{1,\ldots,n\}$.
2. $\forall A, B \in \mathbf{A} : \mathbf{V}_{i,A} \cap \mathbf{V}_{j,B} = \emptyset$*, for each* $\mathbf{V}_{i,A} \in \mathbf{P}_i$ *and* $\mathbf{V}_{j,B} \in \mathbf{P}_j$ *with* $i \neq j \in \{1,\ldots,n\}$.
3. $\forall A \in \mathbf{A} : \mathbf{V}_A = \cup_{1 \leq i \leq n} \mathbf{V}_{i,A}$

Thus, each set $\mathbf{P}_i$ in the collection $\mathbf{P}$ consists of value sets each of which is a subset of values of an attribute. For each attribute $A \in \mathbf{A}$ and according to the second and the third property in Definition 2.2, the sets $\mathbf{V}_{1,A} \in \mathbf{P}_1, \ldots, \mathbf{V}_{n,A} \in \mathbf{P}_n$ define disjoint partitions of $\mathbf{V}_A$. The value set of the partition $\mathbf{P}_i$ is defined as

$$\mathbf{V}_i = \bigcup_{A \in \mathbf{A}} \mathbf{V}_{i,A} \tag{2.7}$$

Since the sets $\mathbf{V}_{i,A}$ $(1 \leq i \leq n)$ are disjoint partitions of $\mathbf{V}_A$ for each $A \in \mathbf{A}$, the sets $\mathbf{V}_i$ $(1 \leq i \leq n)$ define also disjoint partitions of the set $\mathbf{V}$ (see Equations 1.2 and 1.3 in Chapter 1).

Definition 2.3. *(Partitioning function) Let $s_1, \ldots, s_n, S$ be $n+1$ ($n \geq 1$) positive integer numbers with the properties:*

$$s_1 + s_2 + \cdots + s_n = S \tag{2.8}$$

$$s_1 < s_2 < \cdots < s_n \tag{2.9}$$

And let h be a hash-function that can evenly distribute the values of the set $\mathbf{V}$ across S buckets. We define $\delta : \mathbf{V} \rightarrow \{1, 2, \ldots, n\}$ as

$$\delta(v) = \begin{cases} 1 & \textit{if } h(v) \in [0, s_1) \\ i+1 & \textit{if } h(v) \in [\Sigma_{j=1}^{i} s_j, \Sigma_{j=1}^{i+1} s_j) \\ & \quad \textit{for } i = 1, \ldots, n-1 \textit{ and } n > 1. \end{cases} \tag{2.10}$$

and call it a partitioning function. We call $s_1, \ldots, s_n$ the steps of δ.

Thus, our partitioning strategy is a composite of two functions. The first function is the hash-function h, which distributes the values across S buckets. The second function is the aggregation function δ, which aggregates the buckets according the steps s_i ($1 \leq i \leq n$) to create n partitions. Notice that each interval in Equation 2.10 is open on the right. Figure 2.2 illustrates the function δ.

Now, based on the partitioning function δ, we define each set $\mathbf{V}_{i,A}$ in a partition $\mathbf{P}_i \in \mathbf{P} = \{\mathbf{P_1}, \ldots, \mathbf{P_n}\}$ as follows:

$$\mathbf{V}_{i,A} = \{v \mid v \in \mathbf{V}_A \text{ and } \delta(v) = i\} \tag{2.11}$$

Thus, each set $\mathbf{V}_i$ (see Equation 2.7) contains about $(100 \times s_i)/S$ percent of the values of $\mathbf{V}$. Figure 2.3 illustrates the properties of the data partitioning based on the partitioning function δ. Notice that if $n = 1$ ($s_1 = S$), there is no partitioning and δ maps $\mathbf{V}$ to $\mathbf{V}$.

Example 2.3. *For $s_1 = 1, s_2 = 10, s_3 = 20, s_4 = 69$ and $S = 100$, we have $\delta : \mathbf{V} \rightarrow \{1, 2, 3, 4\}$ as*

$$\delta(v) = \begin{cases} 1 & \textit{if } h(v) \in [0, 1) \\ 2 & \textit{if } h(v) \in [1, 11) \\ 3 & \textit{if } h(v) \in [11, 31) \\ 4 & \textit{if } h(v) \in [31, 100) \end{cases}$$

The function δ divides $\mathbf{V}$ into four disjoint partitions $\mathbf{V}_i$ ($1 \leq i \leq 4$), where $\mathbf{V}_1$ contains 1 % of $\mathbf{V}$, $\mathbf{V}_2$ contains 10 % of $\mathbf{V}$, $\mathbf{V}_3$ contains 20 % of $\mathbf{V}$, and $\mathbf{V}_4$ contains 69 % of $\mathbf{V}$.

The next lemma shows how to compute the clusters of the entire dataset $\mathbf{D}$ from the clusters of each of its partitions.

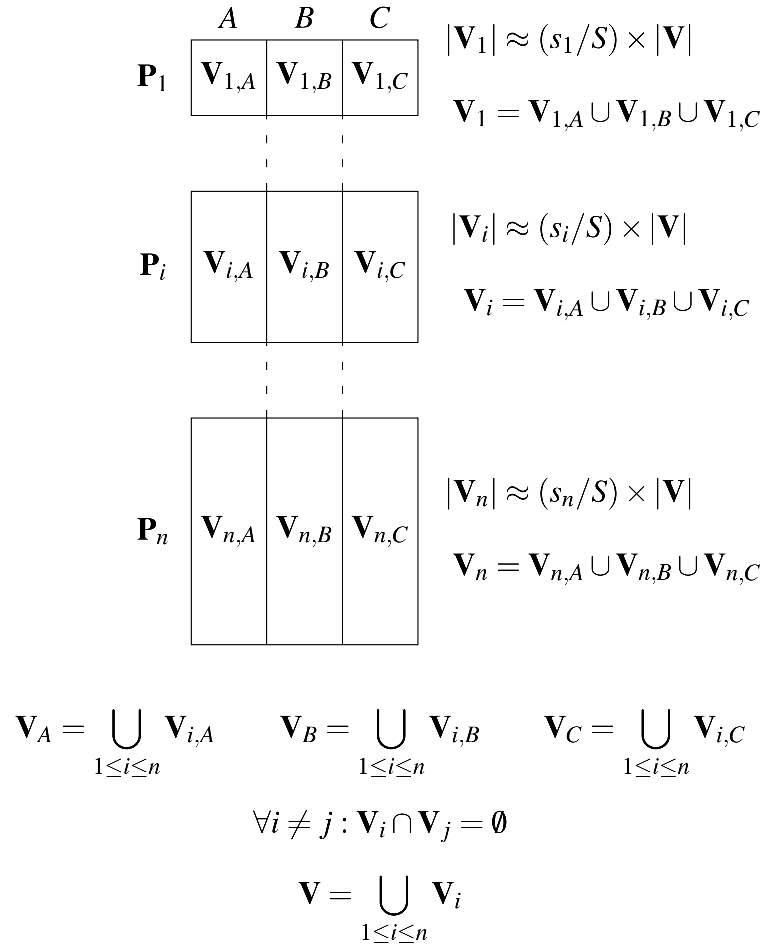


Figure 2.3 Properties of the data partitioning based on the partitioning function δ

Lemma 2.6. *Let* $\mathbf{P} = \{\mathbf{P}_i \mid 1 \leq i \leq n\}$ *be a partitioning of* $\mathbf{D}$*, and let* $\mathbf{AC}_i$ *be the attribute clustering over* $\mathbf{V}_i$ *(*$1 \leq i \leq n$*). Then, the attribute clustering* $\mathbf{AC}$ *over* $\mathbf{V}$ *is*

$$\mathbf{AC} = \bigcup_{1 \leq i \leq n} \mathbf{AC}_i \tag{2.12}$$

Proof. Let $f_i : \mathbf{V}_i \to 2^{\mathbf{A}}$ be the generator of $\mathbf{AC}_i$ ($1 \leq i \leq n$). Since the sets $\mathbf{V}_i$ are disjoint subsets of $\mathbf{V}$, we can define the function $f : \mathbf{V} \to 2^{\mathbf{A}}$ as follows:

$$f(v) = f_i(v) \text{ if } v \in \mathbf{V}_i$$

We have to show that f satisfies Property 2.1 stated in Definition 2.1.

We assume that there is a value v and attribute $A \in \mathbf{A}$ with $v \in \mathbf{V}_A$ and $A \notin f(v)$. Based on the construction of f and on Definition 2.2, it exists $i \in \{1,\ldots,n\}$ for which we have $v \in \mathbf{V}_{i,A} \subseteq \mathbf{V}_i$ and $f(v) = f_i(v)$. According to our assumption, we have $A \notin f_i(v)$ which contradicts the fact that f_i is a generator of $\mathbf{AC}_i$. Thus, our assumption is wrong. In the same way, we can show that there is no attribute $A \in \mathbf{A}$ with $A \in f(v)$ and $v \notin \mathbf{V}_A$. Thus, f satisfies Property 2.1.

Since $\mathbf{V}_i$ ($1 \leq i \leq n$) are disjoint subsets of the set $\mathbf{V}$, we have

$$\begin{aligned}
\mathbf{AC} = f(\mathbf{V}) &= \{\mathbf{C} \subseteq \mathbf{A} \mid \exists v \in \mathbf{V} : f(v) = \mathbf{C}\} \\
&= \{\mathbf{C} \subseteq \mathbf{A} \mid \exists v \in \bigcup_{1 \leq i \leq n} \mathbf{V}_i : f(v) = \mathbf{C}\} \\
&= \bigcup_{1 \leq i \leq n} \{\mathbf{C} \subseteq \mathbf{A} \mid \exists v \in \mathbf{V}_i : f(v) = \mathbf{C}\} \\
&= \bigcup_{1 \leq i \leq n} \{\mathbf{C} \subseteq \mathbf{A} \mid \exists v \in \mathbf{V}_i : f_i(v) = \mathbf{C}\} \\
&= \bigcup_{1 \leq i \leq n} f_i(\mathbf{V}_i) \\
&= \bigcup_{1 \leq i \leq n} \mathbf{AC}_i
\end{aligned}$$

□

Based on the clusters of a partition, the next lemma shows that if an attribute can not be part of any IND in a partition, then this attribute can also not be part of any IND in the entire dataset $\mathbf{D}$.

Lemma 2.7. *If for a partition* $\mathbf{P}_i \in \mathbf{P}$*, there is an attribute* $A \in \mathbf{A}$ *satisfying*

$$\mathbf{I}_{i,A} = \{B \in \mathbf{A} \mid \mathbf{V}_{i,A} \subseteq \mathbf{V}_{i,B}\} = \{A\} \tag{2.13}$$

$$\forall B \in \mathbf{A} \setminus \{A\} : A \notin \mathbf{I}_{i,B} = \{B' \in \mathbf{A} \mid \mathbf{V}_{i,B} \subseteq \mathbf{V}_{i,B'}\} \tag{2.14}$$

then, there is no $B \in \mathbf{A} \setminus \{A\}$ *such that* $A \subseteq B \in \mathbf{I}$ *or* $B \subseteq A \in \mathbf{I}$*, meaning there is no non-trivial valid uIND in* $\mathbf{D}$ *in which* A *occurs.*

Proof. i) An attribute A does not occur on the left-hand-side of any non-trivial uIND if and only if

$$\mathbf{I}_A = \{A\}$$

We have

$$\begin{aligned}
\mathbf{I}_A &= \{B \mid B \in \mathbf{A} \wedge \mathbf{V}_A \subseteq \mathbf{V}_B\} && \text{(Equation 1.9)}\\
&= \bigcap_{\mathbf{C}\in\mathbf{AC}\wedge A\in\mathbf{C}} \mathbf{C} && \text{(Lemma 2.3)}\\
&= \bigcap_{1\leq i\leq|\mathbf{P}|} (\bigcap_{\mathbf{C}\in\mathbf{AC}_i\wedge A\in\mathbf{C}} \mathbf{C}) && \text{(Lemma 2.6)}\\
&= \bigcap_{1\leq i\leq|\mathbf{P}|} \mathbf{I}_{i,A} && \text{(Lemma 2.3)}\\
&= \mathbf{I}_{1,A} \cap \cdots \cap \{A\} \cap \cdots \cap \mathbf{I}_{|\mathbf{P}|,A} && \text{(Condition 2.13)}\\
&= \{A\} && (A \in \mathbf{I}_{j,A}, \forall j = 1, \ldots, |\mathbf{P}|)
\end{aligned}$$

ii) An attribute A does not occur on the right-hand-side of any non-trivial uIND if and only if

$$A \notin \bigcup_{B\in\mathbf{A}\setminus\{A\}} \mathbf{I}_B$$

We have

$$\begin{aligned}
\bigcup_{B\in\mathbf{A}\setminus\{A\}} \mathbf{I}_B &= \bigcup_{B\notin\mathbf{A}\setminus\{A\}} (\bigcap_{\mathbf{C}\in\mathbf{AC}\wedge B\in\mathbf{C}} \mathbf{C}) && \text{(Lemma 2.3)}\\
&= \bigcup_{B\notin\mathbf{A}\setminus\{A\}} (\bigcap_{1\leq i\leq|\mathbf{P}|} (\bigcap_{\mathbf{C}\in\mathbf{AC}_i\wedge B\in\mathbf{C}} \mathbf{C})) && \text{(Lemma 2.6)}\\
&= \bigcup_{B\notin\mathbf{A}\setminus\{A\}} (\mathbf{I}_{1,B} \cap \cdots \cap \mathbf{I}_{i,B} \cap \cdots \cap \mathbf{I}_{|\mathbf{P}|,B}) && \text{(Lemma 2.3)}\\
&\not\supseteq \{A\} && \text{(Condition 2.14)}
\end{aligned}$$

For any $B \in \mathbf{A} \setminus \{A\}$, $A \notin \mathbf{I}_{i,B}$ (Condition 2.14). Thus, $A \notin \mathbf{I}_{1,B} \cap \cdots \cap \mathbf{I}_{i,B} \cap \cdots \cap \mathbf{I}_{|\mathbf{P}|,B}$. □

knowing the clusters over a partition $\mathbf{V}_i$, allow us to detect attributes that can not be part of any INDs in $\mathbf{D}$ so that we can discard them from the process of generating the clusters from all subsequent partitions.

Lemma 2.8. *Let* $\mathbf{I}_{i,A}$ *be the set of all attributes* B *for which* $A \subseteq B$ *is valid in the partition* $\mathbf{P}_i \in \mathbf{P}$*. Then, the set* $\mathbf{I}_A$ *of all attributes* B *for which* $A \subseteq B$ *is valid in* $\mathbf{D}$ *is*

$$\mathbf{I}_A = \bigcap_{1\leq i\leq|\mathbf{P}|} \mathbf{I}_{i,A} \tag{2.15}$$

Proof. Let $\mathbf{A}_i$ be the attribute clustering of $\mathbf{P}_i$. According to Lemma 2.3, we have

$$\mathbf{I}_{i,A} = \{B \mid B \in \mathbf{A} \wedge \mathbf{V}_{i,A} \subseteq \mathbf{V}_{i,B}\} = \bigcap_{\mathbf{C} \in \mathbf{AC}_i \wedge A \in \mathbf{C}} C \tag{2.16}$$

We also have

$$\begin{aligned}
\mathbf{I}_A &= \{B \mid B \in \mathbf{A} \wedge \mathbf{V}_A \subseteq \mathbf{V}_B\} \\
&= \bigcap_{\mathbf{C} \in \mathbf{AC} \wedge A \in \mathbf{C}} C && (\textit{Lemma } 2.3) \\
&= \bigcap_{1 \leq i \leq |\mathbf{P}|} \Big(\bigcap_{\mathbf{C} \in \mathbf{AC}_i \wedge A \in \mathbf{C}} C \Big) && (\textit{Lemma } 2.6) \\
&= \bigcap_{1 \leq i \leq |\mathbf{P}|} \mathbf{I}_{i,A} && (\textit{Equation } 2.16)
\end{aligned}$$

□

2.4 S-INDD++

S-INDD++ consists of the following main components: Partitioning the dataset, preparing the partitions, generating the clusters from the partitions, and then deriving uINDs from the generated clusters. The goal of the preparation of the partitions is to bring the content of each partition into the format required by S-INDD for generating the attribute clustering from each partition. The subsection below describes the workflow between these components, while the partitioning, the preparation of a partition, and the generation of the clusters from a partition are described in Subsections 2.4.2, 2.4.3, and 2.4.4, respectively.

2.4.1 Overall workflow

In the first step, S-INDD++ (see Algorithm 2) divides the dataset $\mathbf{D}$ according to the partitioning function δ (see Definition 2.3) by calling Algorithm 3 (Line 2), which returns an ordered list of the computed partitions. The partitions in the returned list is ordered according to partition number $i \in \{1, \ldots, n\}$. Notice that a partition with the index i contains all values $v \in \mathbf{V}$ for which $\delta(v) = i$ (see Equations 2.7 and 2.11). S-INDD++ processes the partitions according to the ascending order of their numbers (Lines 3-20), which is essential to save computation time by firstly discarding attributes from smaller partitions. Discarding attributes from the discovering process after handing a partition is based on the application of Lemma 2.7 (Lines 11-19).

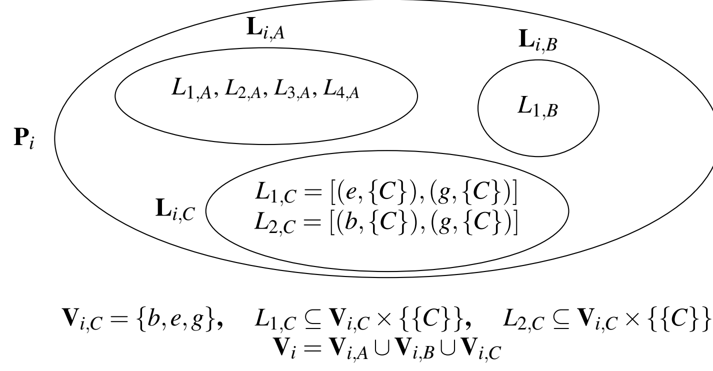


Figure 2.4 Partition layout

Algorithm 2: S-INDD++

Input : $\mathbf{R}$, $\mathbf{D}$, $\mathbf{A}$, n, k, δ

Output : $\mathbf{I}$

1 $\mathbf{AC} \leftarrow \emptyset$; $\mathbf{A}^- \leftarrow \emptyset$; $\mathbf{I} \leftarrow \emptyset$

2 $[\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n] \leftarrow$ `computePartitions(`$\mathbf{R}$, $\mathbf{D}$, $\mathbf{A}$, n, δ`)`

3 **for** i = *1* **to** n **do**

4   **foreach** $A \in \mathbf{A}^-$ **do**

5         $\mathbf{P}_i \leftarrow \mathbf{P}_i \setminus \mathbf{L}_{i,A}$

6   **foreach** $\mathbf{L}_{i,A} \in \mathbf{P_i}$ **do**

7   **if** $|\mathbf{L}_{i,A}| > 1$ **then**

8   `mergeLists(`$\{L_{i,A} \mid L_{i,A} \in \mathbf{L}_{i,A}\}$`)`

9     $\mathbf{AC}_i \leftarrow$ S-INDD ($\mathbf{P}_i$, $\mathbf{A}$, k)

10     $\mathbf{AC} \leftarrow \mathbf{AC} \cup \mathbf{AC}_i$

11     $\mathbf{RH} \leftarrow \emptyset$

12   **foreach** $A \in \mathbf{A}$ **do**

13         $\mathbf{I}_{i,A} \leftarrow \cap_{\mathbf{C} \in \mathbf{AC}_i \wedge A \in \mathbf{C}} \mathbf{C}$

14         $\mathbf{I}_{i,A} \leftarrow \mathbf{I}_{i,A} \setminus \{A\}$

15         $\mathbf{RH} \leftarrow \mathbf{RH} \cup \mathbf{I}_{i,A}$

16   **foreach** $A \in \mathbf{A}$ **do**

17   **if** $\mathbf{I}_{i,A} = \emptyset \wedge A \notin \mathbf{RH}$ **then**

18             $\mathbf{A}^- \leftarrow \mathbf{A}^- \cup \{A\}$

19             $\mathbf{A} \leftarrow \mathbf{A} \setminus \{A\}$

20   **if** $\mathbf{A} = \emptyset$ **then break**

21 **if** $\mathbf{A} \neq \emptyset$ **then**

22   **foreach** $A \in \mathbf{A}$ **do**

23         $\mathbf{I}_A \leftarrow \cap_{\mathbf{C} \in \mathbf{AC} \wedge A \in \mathbf{C}} \mathbf{C}$

24         $\mathbf{I} \leftarrow \mathbf{I} \cup \{A \subseteq B \mid B \in \mathbf{I_A}\}$

Figure 2.4 illustrates the partition layout as designed by Algorithm 3. Each partition $\mathbf{P}_i$ contains a set of lists denoted as $\mathbf{L}_{i,A}$ for each attribute $A \in \mathbf{A}$. Each list $L_{i,A} \in \mathbf{L}_{i,A}$ is a *subset* of $\mathbf{V}_{i,A} \times \{\{A\}\}$ *sorted* according to the values in $\mathbf{V}_{i,A}$ and stored as a file in a repository representing $\mathbf{P}_i$.

Handling a partition $\mathbf{P}_i$ finishes with finding each attribute that can not be a part of any valid IND in $\mathbf{D}$ by applying Lemma 2.7 on the attribute clustering $\mathbf{AC}_i$ over $\mathbf{V}_i$ (Lines 11-19). We call such an attribute a discarded (or pruned) attribute and denote the set of those attributes by $\mathbf{A}^-$. We also call an attribute that has not been discarded a survived (or an active) attribute. The set $\mathbf{A}$ contains only the active attributes. After generating the attribute clustering $\mathbf{AC}_i$ over $\mathbf{V}_i$, S-INDD++ computes the set $\mathbf{I}_{i,A}$ for every survived attribute based on Lemma 2.3 (Line 13). The set $\mathbf{I}_{i,A}$ contains every survived attribute B, for which $A \subseteq B$ is valid in $\mathbf{P}_i$. S-INDD++ removes A from $\mathbf{I}_{i,A}$ to make sure that the trivial IND $A \subseteq A$ will not be considered (Line 14). Then, S-INDD++ adds A to the set of discarded attributes $\mathbf{A}^-$ if (i) the set $\mathbf{I}_{i,A}$ is empty implying that A can not occur on the left-hand-side of any valid IND in $\mathbf{D}$, and (ii) the set $\mathbf{RH}$ does not contain A implying that A can not occur on the right-hand.side of any valid IND in $\mathbf{D}$ (Lines 16-19). The set $\mathbf{RH}$ contains every attribute that occurs on the right-hand-side of a valid IND in the partition $\mathbf{P}_i$ (Line 15).

Handling a partition $\mathbf{P}_i$ starts with removing the sets $\mathbf{L}_{i,A}$ of the discarded attributes $A \in \mathbf{A}^-$ from the partition (Lines 4-5). After that, for each survived $A \in \mathbf{A}$, the lists contained in $\mathbf{L}_{i,A}$ have to be merged by calling Algorithm 4 to become one list L_A that consists of all tuples $(v, \{A\}) \in \mathbf{V}_i \times \{\{A\}\}$ sorted according to v's (Lines 6-8). Algorithm 4 replaces the set $\mathbf{L}_{i,A}$ by the new list L_A, meaning that after finishing the work of Algorithm 4, $\mathbf{P}_i$ contains only one list L_A for each attribute A being still active. For instance, the set $\mathbf{L}_{i,C}$ in Figure 2.4 will be replaced by the list $L_C = [(b, \{C\}), (e, \{C\}), (g, \{C\})]$ resulting from merging $L_{1,C}$ and $L_{2,C}$ by Algorithm 4. Notice if the set $\mathbf{L}_{i,A}$ contains only one list $L_{1,A}$, then there will be no merge and $L_{1,A}$ will implicitly be renamed to L_A. For instance, there is no merge for $\mathbf{L}_{i,B}$ in Figure 2.4 and the list $L_{1,B}$ will be renamed to L_B.

After processing $\mathbf{P}_i$ by Algorithm 4, $\mathbf{P}_i$ is in the format required by S-INDD for generating the clusters. Thus, S-INDD++ computes the attribute clustering $\mathbf{AC}_i$ over $\mathbf{V}_i$ by applying S-INDD on $\mathbf{P}_i$ (Line 9). The desired attribute clustering $\mathbf{AC}$ over $\mathbf{V}$ is computed from $\mathbf{AC}_i$ $(1 \leq i \leq n)$ based on Lemma 2.6 (Line 10). Having the attribute clustering $\mathbf{AC}$ generated from the partitions, S-INDD++ computes the set $\mathbf{I}$ from the clusters by applying Lemma 2.3 (Lines 21-24).

2.4.2 Computing the partitions

Algorithm 3 describes the partitioning process, which requires the relations $\mathbf{R}$ and their instances (tables) in $\mathbf{D}$, th attribute set $\mathbf{A}$, the number of partitions n, and the partitioning function δ (see Definition 2.3) as input parameters. At the beginning, Algorithm 3 creates n repositories on the hard disk, where each repository represents one partition $\mathbf{P}_i \in \mathbf{P}$ $(1 \leq i \leq n)$ of the dataset $\mathbf{D}$ (Line 1).

Algorithm 3 divides the values of the input dataset by iterating the dataset relation-wise in order to keep possibly all partitions for one relation instance at a time in main memory (Lines 2-28). For each relation instance r, Algorithm 2.4.2 reads the values in a tuple-wise manner (Lines 8-21). From each

Algorithm 3: Computation of the partitions

Input : **R**, **D**, **A**, n, δ
Output : **P**

```
1  P ← createRepositories (n)
2  foreach R ∈ R do
3      array ValSets size n × |R| as Set
4      array SetCounts size n × |R| as Integer
5      foreach (i,A) ∈ {1,...,n} × R do
6          ValSets[i][A] ← ∅
7          SetCounts[i][A] ← 0
8      foreach t ∈ r do
9          foreach A ∈ R do
10             v ← t[A]
11             i ← δ(v)
12             ValSets[i][A] ← ValSets[i][A] ∪ {v}
13             if memoryExhausted() then
14                 foreach (i,A) ∈ {1,...,n} × R do
15                     if ValSets[i][A] ≠ ∅ then
16                         SetCounts[i][A] ← SetCounts[i][A] + 1
17                         j ← SetCounts[i][A]
18                         L_{j,A} ← ValSets[i][A] × {{A}}
19                         sortList (L_{j,A})
20                         write (L_{j,A}, P[i])
21                         ValSets[i][A] ← ∅
22     foreach (i,A) ∈ {1,...,n} × R do
23         if ValSets[i][A] ≠ ∅ then
24             SetCounts[i][A] ← SetCounts[i][A] + 1
25             j ← SetCounts[i][A]
26             L_{j,A} ← ValSets[i][A] × {{A}}
27             sortList (L_{j,A})
28             write (L_{j,A}, P[i])
```

tuple t, it fetches all those values that belong to the attributes of the corresponding relation (Line 10). The partition number i ($1 \leq i \leq n$) of each value is calculated by applying the partitioning function δ (Line 11). Notice that if δ places a value v into a partition $\mathbf{P}_i$, then it places all values u with $u = v$ into the same partition $\mathbf{P}_i$ (see Definition 2.3), which guarantees the pairwise disjunction of the value sets $\mathbf{V}_i$ ($1 \leq i \leq n$), each of which is the value set of one partition $\mathbf{P}_i$ (see Equation 2.7).

Every time Algorithm 3 partitions a tuple, it checks the consumption of the main memory (Lines 13-21). If the main memory is exhausted (e.g., if less than 15% of the memory is free), Algorithm 3 writes the current value set $\mathbf{V}_i$, which is equal $\cup_{A \in R} ValSets[i][A]$, into the repository of $\mathbf{P}_i$. Writing the value set $\mathbf{V}_i$ corresponds to writing a sorted list of tuples $(v, \{A\})$ of the set $\mathbf{V}_{i,A} \times \{\{A\}\}$ into the repository of $\mathbf{P}_i$ for each attribute A, for which $ValSets[i][A]$ is not empty (Lines 15-21).

Because of memory exhaustion, Algorithm 3 writes the sets $\mathbf{V}_{i,A}$ ($1 \leq i \leq n$ and $A \in \mathbf{A}$) in more than one stage, causing that each partition $\mathbf{P}_i$ may contain more than one list for each attribute. We denote these lists with $\mathbf{L}_{i,A}$. To distinguish between the lists in $\mathbf{L}_{i,A}$, Algorithm 3 manages a counter $SetCounts[i][A]$ for each pair $\mathbf{P}_i \times A$. This counter has to be increased each time Algorithm 3 has to write a list of a subset of the set $\mathbf{V}_{i,A} \times \{\{A\}\}$ into the repository of the partition with the index i (Line 16). After dividing all tuples, Algorithm 3 writes (flushes) the remaining contain of the main memory into the repositories (Lines 22-28).

Example 2.4. *Let $R = [A, B]$ be a relational schema and let*

$$r = \{t_1 = (c, z), t_2 = (a, x), t_3 = (c, x), t_4 = (b, x)\}$$

be an instance of R. We assume $n = 1$, meaning that there is only one partition $\mathbf{P}_1$. If the main memory is exhausted after processing the first three tuples t_1, t_2, and t_3, we have $SetCounters[1][A] = 1$ and $SetCounters[1][B] = 1$ and $\mathbf{P}_1$ will contain

$$L_{1,A} = [(a, \{A\}), (c, \{A\})] \qquad L_{1,B} = [(x, \{B\}), (z, \{B\})]$$

After processing the last tuple t_4, we have $SetCounters[1][A] = 2$ and $SetCounters[1][B] = 2$, and $\mathbf{P}_1$ will be updated to contain

$$L_{1,A} = [(a, \{A\}), (c, \{A\})] \qquad L_{1,B} = [(x, \{B\}), (z, \{B\})]$$
$$L_{2,A} = [(b, \{A\})] \qquad L_{2,B} = [(x, \{B\})]$$

Thus, $\mathbf{P}_1$ consists of two groups of lists:

$$\mathbf{L}_{1,A} = \{L_{1,A}, L_{2,A}\} \qquad \mathbf{L}_{1,B} = \{L_{1,B}, L_{2,B}\}$$

After processing each of both by Algorithm 4 in Subsection 2.4.3 below, $\mathbf{P}_i$ contains only

$$L_A = [(a, \{A\}), (b, \{A\}), (c, \{A\})] \qquad L_B = [(x, \{B\}), (z, \{B\})]$$

Algorithm 4: Mergeing the lists of $\mathbf{L}_{i,A}$

```
   Input  : L_{1,A}, ..., L_{j,A}, ..., L_{m,A}
   Output : L_A
1  Queue ← createPriorityQueue(L_{1,A}, ..., L_{m,A})
2  L_A ← []
3  writer ← createWriter(L_A)
4  while Queue ≠ ∅ do
5      reader ← Queue·pull()
6      (v, {A}) ← reader·readCurrent()
7      L_A ← L_A + [(v, {A})]
8      if memoryExhausted() then
9          writer·write (L_A)
10         L_A ← []
11     if reader·hasNext() then
12         reader·readNext()
13         Queue·add(reader)
14     while Queue ≠ ∅ do
15         reader ← Queue·peek()
16         (v′, {A}) ← reader·readCurrent()
17         if v ≠ v′ then
18             break
19         else
20             reader ← Queue·pull()
21             if reader·hasNext() then
22                 reader·readNext()
23                 Queue·add(reader)
24 writer·write(L_A)
25 writer·close()
```

2.4.3 Postprocessing of a partition

After finishing the partitioning process, each partition $\mathbf{P}_i$ ($1 \leq i \leq n$) contains sets $\mathbf{L}_{i,A}$ ($A \in \mathbf{A}$). Each set $\mathbf{L}_{i,A}$ consists of m ($1 \leq m$ is $ValSets[i][A]$ after finishing Algorithm 3) lists $L_{j,A}$ ($1 \leq j \leq m$), where each list $L_{j,A}$ contains *a subset* of tuples $(v, \{A\})$ of the set $\mathbf{V}_{i,A} \times \{\{A\}\}$ *sorted* according to the values $v \in \mathbf{V}_{i,A}$. The purpose of Algorithm 4 is to merge the lists in $\mathbf{L}_{i,A}$ to produce one list L_A containing *all* elements of $\mathbf{V}_{i,A} \times \{\{A\}\}$. Notice that each $L_{j,A}$ ($1 \leq j \leq m$) is represented as a file saved in the repository of $\mathbf{P}_i$.

Algorithm 4 is an adoption of the external merge-sort algorithm. For every value v, it collects v's tuple $(v, \{A\})$ simultaneously from all lists $L_{j,A}$, in each of which $(v, \{A\})$ occurs, to write it again into the new list L_A. To achieve this idea, Algorithm 4 associates every list (file) $L_{j,A} \in \mathbf{L}_{i,A}$ ($1 \leq j \leq m$) with a sequential file reader. It opens all of these readers at once and manages them using a priority

queue (Line 1). For any two readers *reader 1* and *reader 2*, *reader 1* has a higher priority in the queue than *reader 2* if the value v in the tuple $(v, \{A\})$ is smaller than the value v' in $(v', \{A\})$, where $(v, \{A\})$ is the entry that *reader 1* can currently read and $(v', \{A\})$ is the entry that *reader 2* can currently read. Every time Algorithm 4 has to read a tuple, it pulls a reader from the queue, reads the tuple to which the pulled reader currently points, and then checks whether the corresponding list still has tuples that have not been read yet. In the positive case, the reader will be added again to the queue after moving it to point at the next tuple in the associated list (Lines 11-13 and Lines 20-23).

After reading a tuple $(v, \{A\})$ for the first time, Algorithm 4 adds it to the output list L_A (Lines 6-7). Then, it checks the consumption of the main memory (Lines 8-10). If the main memory is exhausted, the current content of L_A will be written (flushed) into the output file, to which L_A has been associated in Line 3. Since the generation of the clusters by S-INDD requires that L_A must be duplicate free, meaning that $(v, \{A\})$ ($\forall v \in \mathbf{V}_i$) must not occur than once in L_A, Algorithm 4 skips adding $(v, \{A\})$ again to L_A by moving the reader of each list containing $(v, \{A\})$ to point at the next different tuple (Lines 14-23).

2.4.4 Generating the clusters

S-INDD presented in Algorithm 5 computes the attribute clustering $\mathbf{AC}_i$ for the partition $\mathbf{P}_i$ of the dataset $\mathbf{D}$. S-INDD basically computes the function f defined in the proof of Lemma 2.1 for every value v by gradually aggregating all attributes whose value sets contain that value v. Notice that $\mathbf{P}_i$ is the partition that is currently processed by S-INDD++.

The gradual computation of the generator f is achieved in two stages. The first stage consists of a sequence of aggregate operations (Lines 1-2) resulting in the computation of a large part of f, meaning that after finishing the aggregation stage and for at least one value v, $f(v)$ may still not be the maximum set of attributes whose value sets contain the value v. The second stage, therefore, completes the computation of f and then generates the clusters (Lines 3-8).

Initially, $\mathbf{P}_i$ contains $|\mathbf{A}|$ lists. Every initial list $L_A \in \mathbf{P}_i$ relates to an attribute $A \in \mathbf{A}$ and its elements are all elements of the set $\mathbf{V}_{i,A} \times \{\{A\}\}$ sorted according to the values in $\mathbf{V}_{i,A}$. The partition $\mathbf{P}_i$ is presented to S-INDD as an external repository on the hard disk in which every list L_A ($A \in \mathbf{A}$) is saved as a file. Example 2.5 demonstrates the initial content of $\mathbf{P}_i$.

Example 2.5. *For* $\mathbf{V}_{i,A} = \{a,b,d\}$, $\mathbf{V}_{i,B} = \{a,b,c,d\}$, $\mathbf{V}_{i,C} = \{e,f\}$, *and* $\mathbf{V}_{i,D} = \{a,c\}$, $\mathbf{P}_i$ *initially contains the following lists:*

$$L_A = [(a,\{A\}),(b,\{A\}),(d,\{A\})] \qquad L_B = [(a,\{B\}),(b,\{B\}),(c,\{B\}),(d,\{B\})]$$
$$L_C = [(e,\{C\}),(f,\{C\})] \qquad L_D = [(a,\{D\}),(c,\{D\})]$$

Algorithm 5: S-INDD

Input : $\mathbf{P}_i$, $\mathbf{A}$, k
Output : $\mathbf{AC}_i$

1 **while** *($\mathbf{P}_i$ contains k or more than k lists)* **do**
2   `aggregateLists(`$\mathbf{P}_i$, k`)`

3 $Queue \leftarrow$ `createPriorityQueue(`$\mathbf{P}_i$`)`
4 $\mathbf{AC}_i \leftarrow \emptyset$
5 **while** $Queue \neq \emptyset$ **do**
6   $(v, \mathbf{AS}) \leftarrow$ `collectNextAttSets(`$Queue$`)`
7   $\mathbf{C} \leftarrow \bigcup_{\mathbf{A}^v \in \mathbf{AS}} \mathbf{A}^v$
8   $\mathbf{AC}_i \leftarrow \mathbf{AC}_i \cup \{\mathbf{C}\}$

Algorithm 6: Aggregation of k lists

Input : $\mathbf{P}_i$, k

1 $L_1, L_2, \ldots, L_k \leftarrow$ `selectLists(`$\mathbf{P}_i, k$`)`
2 $Queue \leftarrow$ `createPriorityQueue(`$L_1, L_2, \ldots, L_k$`)`
3 $L \leftarrow [\,]$
4 **while** $Queue \neq \emptyset$ **do**
5   $(v, \mathbf{AS}) \leftarrow$ `collectNextAttSets(`$Queue$`)`
6   $\mathbf{C} \leftarrow \bigcup_{\mathbf{A}^v \in \mathbf{AS}} \mathbf{A}^v$
7   $L \leftarrow L + [(v, \mathbf{C})]$

8 $\mathbf{P}_i \leftarrow \mathbf{P_i} \setminus \{L_1, \ldots, L_k\}$
9 `write(`$L, \mathbf{P}_i$`)`

Algorithm 7: Collection of the next attribute set

Input : $Queue$
Output : $(v, \mathbf{AS})$

1 $\mathbf{AS} \leftarrow \emptyset$
2 **repeat**
3   $fr \leftarrow Queue$ $\cdot$`pull()`
4   $(v, \mathbf{A}^v) \leftarrow fr\cdot$`readCurrent()`
5   $(v, \mathbf{AS}) \leftarrow (v, \mathbf{AS} \cup \{\mathbf{A}^v\})$
6   **if** $fr\cdot$`hasNext()` **then**
7     $fr\cdot$`readNext()`
8     $Queue\cdot$`add(`fr`)`
9   **if** $Queue \neq \emptyset$ **then**
10     $fr \leftarrow Queue$ $\cdot$`peek()`
11     $(v', \mathbf{A}^{v'}) \leftarrow fr\cdot$`readCurrent()`
12 **until** $(Queue = \emptyset)$ *or* $(v' \neq v)$

Aggregation of lists The result of the aggregate operation (see Algorithm 6) is always a sorted list whose elements are a subset of the set

$$\{(v, \mathbf{A}^v) \mid \mathbf{A}^v \neq \emptyset \wedge \mathbf{A}^v \subseteq \cup_{A \in \mathbf{A} \wedge v \in \mathbf{V}_{i,A}} \{A\}\}$$

In details, the aggregate operation reads k ($2 \leq k \leq |\mathbf{A}|$) lists

$$\begin{aligned} L_1 &= [(v_{11}, \mathbf{B}^{v_{11}}) \ldots, (v_{1l_1}, \mathbf{B}^{v_{1l_1}})] \\ &\vdots \\ L_p &= [(v_{p1}, \mathbf{B}^{v_{p1}}) \ldots, (v_{pl_p}, \mathbf{B}^{v_{pl_p}})] \\ &\vdots \\ L_k &= [(v_{k1}, \mathbf{B}^{v_{k1}}), \ldots, (v_{kl_k}, \mathbf{B}^{v_{kl_k}})] \end{aligned}$$

from $\mathbf{P}_i$ and then replaces them with the new list

$$L = [(v_1, \mathbf{A}^{v_1}), (v_2, \mathbf{A}^{v_2}), \ldots, (v_l, \mathbf{A}^{v_l})]$$

that satisfies the following conditions:

$$\begin{aligned} v_1 &= \min_{\substack{1 \leq p \leq k \\ 1 \leq q \leq l_p}} \{v_{pq}\}, & \mathbf{A}^{v_1} &= \bigcup_{\substack{1 \leq p \leq k \\ 1 \leq q \leq l_p}} \mathbf{B}^{v_1} \\ &\vdots \\ v_s &= \min_{\substack{1 \leq p \leq k \\ 1 \leq q \leq l_p}} \{v_{pq} \mid v_{pq} \notin \{v_1, \ldots, v_{s-1}\}\}, & \mathbf{A}^{v_s} &= \bigcup_{\substack{1 \leq p \leq k \\ 1 \leq q \leq l_p}} \mathbf{B}^{v_s} \\ &\text{with } s = 2, \ldots, l \end{aligned}$$

The new list L is duplicate free and sorted according to the values $v_s \in \{v_{pq} \mid 1 \leq p \leq k, 1 \leq q \leq l_p\}$ ($1 \leq s \leq l$). Every set $\mathbf{A}^{v_s}$ in L is the union of all sets $\mathbf{B}^{v_s}$ in the tuples $(v_s, \mathbf{B}^{v_s})$ that occur in the k lists.

S-INDD repeats the aggregate operation (Lines 1-2) until the repository $\mathbf{P}_i$ has fewer than k lists where every new list generated by the aggregate operation has to be stored in the repository $\mathbf{P}_i$ as a temporary result (Line 9 in Algorithm 6). Example 2.6 illustrates the aggregate operation.

Example 2.6. *Based on Example 2.5 and for* $\underline{k = 2}$*,* S-INDD *executes three aggregate operations.*

Selecting L_A *and* L_B *by the first aggregate operation, generates the following list*

$$L_{A,B} = [(a, \{A, B\}), (b, \{A, B\}), (c, \{B\}), (d, \{A, B\})]$$

and changes the content of $\mathbf{P}_i$ *to* $L_{A,B}, L_C,$ *and* L_D.
Selecting $L_{A,B}$ *and* L_D *by the second aggregate operation, produces the list*

$$L_{A,B,D} = [(a,\{A,B,D\}),(b,\{A,B\}),(c,\{B,D\}),(d,\{A,B\})]$$

and changes the content of $\mathbf{P}_i$ *to* $L_{A,B,D}$ *and* L_C.
The last aggregate operation takes the two remaining lists $L_{A,B,D}$ *and* L_C *and replaces them with*

$$L_{A,B,D,C} = [(a,\{A,B,D\}),(b,\{A,B\}),(c,\{B,D\}),(d,\{A,B\}),(e,\{C\}),(f,\{C\})]$$

Thus, after finishing the three aggregate operations, the content of $\mathbf{P}_i$ *consists of only the list* $L_{A,B,D,C}$.

In the case of $\underline{k=3}$, S-INDD *has to execute only one aggregation operation. If the first three lists* $L_A, L_B,$ *and* L_C *are selected for aggregation by Algorithm 6, the list*

$$L_{A,B,C} = [(a,\{A,B\}),(b,\{A,B\}),(c,\{B\}),(d,\{A,B\}),(e,\{C\}),(f,\{C\})]$$

will be generated and the repository $\mathbf{P}_i$ *will be changed to contain only the lists* $L_{A,B,C}$ *and* L_D.

For an efficient implementation of the aggregate operation and for managing a simultaneous reading of k lists (files) from the repository $\mathbf{P}_i$, a priority queue is used by Algorithm 6 (and also by Algorithm 7 - see below). The queue manages k readers (sequential file readers). As explained in Subsection 2.4.3, every reader is associated with a list and points to the entry that can currently be read from the list. For every two file readers fr, fr', reader fr has a higher priority than fr' if and only if the value v in $(v,\mathbf{A}^v)$ is smaller than or equal to the value v' in $(v',\mathbf{A}^{v'})$ where $(v,\mathbf{A}^v)$ is the entry that fr can currently read and $(v',\mathbf{A}^{v'})$ is the entry that fr' can currently read.

The purpose of using a priority queue is to enable an efficient collecting of all sets $\mathbf{A}_1^v,\ldots,\mathbf{A}_{l_v}^v$ $(1 \leq l_v \leq k)$ by a simultaneous and sequential reading of k lists where v is the smallest value among all values that have not been read from the k lists in the queue yet. That is possible in a simultaneous sequential reading because the lists are sorted according to the values $v \in \mathbf{V}$ and the priority in the queue is defined according to the ascending order of the values.

Calculating the clusters After finishing the aggregations, S-INDD generates the attribute clustering $\mathbf{AC}_i$ from all remaining k' $(1 \leq k' < k)$ lists (Lines 4-8).

For every value v, there are still k_v $(1 \leq k_v \leq k')$ lists $L_1,\ldots,L_j,\ldots,L_{k_v}$ containing entries of the form $(v,\mathbf{A}^v)$. S-INDD collects all these remaining entries by calling Algorithm 7 in Line 6. Then, S-INDD computes the set

$$\mathbf{C} = \bigcup_{\substack{1 \leq j \leq k_v \\ (v,\mathbf{A}^v) \in L_j}} \mathbf{A}^v$$

and adds it as a cluster to the set $\mathbf{AC}_i$. Example 2.7 illustrates these steps.

Example 2.7. *According to Example 2.6 and for* $\underline{k=3}$*,* $\mathbf{P}_i$ *contains the lists*

$$L_{A,B,C} = [(a,\{A,B\}),(b,\{A,B\}),(c,\{B\}),(d,\{A,B\}),(e,\{C\}),(f,\{C\})]$$
$$L_D = [(a,\{D\}),(c,\{D\})]$$

after finishing the aggregations.

For the value $v = a$*, there are two entries:* $(a,\{A,B\})$ *in* $L_{A,B,C}$ *and* $(a,\{D\})$ *in* L_D*. Therefore, Algorithm 5 collects the two sets* $\{A,B\}$ *and* $\{D\}$ *by calling algorithm 7 in the first run of the* while-*loop which delivers the tuple:* $(a,\{\{A,B\},\{D\}\})$*. The first cluster is then* $\mathbf{C}_1 = \{A,B\} \cup \{D\} = \{A,B,D\}$ *and consequently* $\mathbf{AC}_i = \{\{A,B,D\}\}$*.*

After the second run of the while-*loop, we have* $\mathbf{AC}_i = \{\{A,B,D\},\{A,B\}\}$*. Calling Algorithm 7 in the third run of the* while-*loop delivers the tuple:* $(c,\{\{B\},\{D\}\})$*. Consequently,* $\mathbf{AC}_i$ *will be extended to* $\mathbf{AC}_i = \{\{A,B,D\},\{A,B\},\{B,D\}\}$*. Computing* $\mathbf{AC}_i$ *finishes after the sixth run of the* while-*loop resulting in* $\mathbf{AC}_i = \{\{A,B,D\},\{A,B\},\{B,D\},\{C\}\}$*.*

2.5 Experimental evaluation

We now present the results of exhaustive experiments carried out to answer the following questions:

1. How effective is inferring the uINDs from the attribute clustering in terms of the reduction of the intersections needed for generating them from the inverted index?

2. How effective is S-INDD++'s partitioning strategy in comparison to that of BINDER? The comparison has to be conducted in terms of the number of candidates that each of both algorithm prunes after processing each partition of its strategy.

3. How significantly does S-INDD++ outperform BINDER when applying each of both algorithm to different real-word datasets?

4. How significantly does S-INDD++ outperform S-INDD? The purpose of this question to know how reducing the time of completely sorting the set values of those attributes that are not a part of any uIND improve the performance?

5. How significantly does S-INDD improve the scalability of SPIDER?

2.5.1 Setup

Datasets Table 2.2 and Table 2.3 present the datasets used in the experiments. Table 2.2 shows some characteristics of datasets in size less than one GB, while Table 2.3 also shows the same characteristics but of datasets in size greater than 11 GB. The first column gives the names of datasets, while their sizes are given in the second column. The third column states the number of non-empty relations (tables) in the corresponding dataset. The total number of attributes (columns) in each dataset is given

Table 2.2 Characteristics of datasets used in the experiments (Size in MB)

D	Size	$\|\mathbf{D}\|$	$\|\mathbf{A}\|$	$\Sigma\|r_i\|$	$\min\|r_i\|$	$\max\|r_i\|$	$\Sigma\|r_i\|/\|\mathbf{D}\|$	$\|\mathbf{I}\|$
SCOP	16	4	22	342,195	11,597	165,299	85,548	43
WIKIPEDIA	540	2	14	14,802,104	777,676	14,024,428	7,401,052	2
BIOSQL	560	15	77	8,306,268	1	1,854,789	553,751	112
LOD	830	13	164	4,792,549	771	1,745,873	368,657	298
ENSEMBL	835	20	130	12,599,658	5	4,339,917	629,982	340
CATH	907	4	25	152,652	1793	67,054	38,163	62

Table 2.3 Characteristics of larger datasets used in the experiments (Size in GB)

D	Size	$\|\mathbf{D}\|$	$\|\mathbf{A}\|$	$\Sigma\|r_i\|$	$\min\|r_i\|$	$\max\|r_i\|$	$\Sigma\|r_i\|/\|\mathbf{D}\|$	$\|\mathbf{I}\|$
H-GENOME	11,2	62	387	116,227,014	4	26,552,339	1,874,629	4995
MB	26,8	178	1053	222,019,536	1	37,212,456	1,247,300	44,803
PDB	44	117	1791	266,352,038	1	218,948,441	2,276,513	35,642
PLISTA	61	4	140	109,669,418	263,767	101,305,267	27,417,354	381

in the fourth column, while the total number of tuples (rows) is sated in the fifth column. The columns labeled with $\min|r_i|$, $\max|r_i|$, and $\Sigma|r_i|/|\mathbf{D}|$ are the minimum, the maximum, and the average number of tuples per relation. The last column states the number of uINDs in the corresponding dataset. For each dataset, we did not count the empty relations, the empty attributes, and the trivial uINDs.

SCOP, BIOSQL, CATH, ENSEMBL, and PDB are all excerpts from biological databases on proteins, dna, and genomes. WIKIPEDIA is a dataset crawled from the Wikipedia knowledge base, and contains page statistics. LOD is an excerpt of linked open data on famous persons and stores many RDF-triples in relational formats. PLISTA Kille et al. [2013] contains anonymized web-log data provided by the advertisement company Plista. H-GENOME is a genome dataset of homo sapiens available at http://www.ensembl.org. MUSICBRAINZ (MB) is an open music encyclopedia that collects music metadata and makes them available to the public at https://musicbrainz.org.

Experimental conditions We performed the experiments on a Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We installed Oracle 11g on the same machine as the database server and used an external disk for the storage of all used datasets. We implemented S-INDD++ and S-INDD in Java 8. We used the open source implementation of BINDER available at github.

2.5.2 The effectiveness of the attribute clustering

To show the effectiveness of the attribute clustering, we compared the number of intersection operations needed to derive the uINDs from the clusters with the number of intersection operations needed to derive them from the inverted index. Table 2.4 presents the results of these comparisons. The last column in this table shows the percentage of the reduction in the number of intersections gained by deriving the uINDs from the attribute clustering. As we can see, the gained reduction is more than 99.6 % for 70 % of the datasets, while the reduction is more than 95.5 % for PDB, more than 92.3 % H-GENOME, and more than 69 % for MB.

In addition, we calculated some statistics about the cluster sizes of the datasets. Table 2.5 presents these statistics. The second column of this table presents the number of the clusters. The columns labeled with $\min|\mathbf{C}_i|$, $\max|\mathbf{C}_i|$, and $\text{median}|\mathbf{C}_i|$ are the minimum cluster size, the maximum cluster size, the median cluster size of the corresponding attribute clustering. The average cluster size is given in the sixth column, while the standard deviation of the average size is given in the last column. As we can see, for instance for the dataset MB, there is at least one value shared by 362 attributes (i.e., about 34.4 % of the attributes). Furthermore, in average each group of 40 attributes (i.e., about 3.8 % of the attributes in this dataset) has at least one value in common. As another example is the dataset H-GENOME in which in average each group of 21 attributes (i.e., about 5.5 % of the attributes) shares at least one value.

Table 2.4 Comparing the number of intersections needed when inferring uINDs from the inverted index (#i-idx-∩) with number of intersections needed when inferring uINDs from the attribute clustering (#**AC**-∩)

D	#i-idx-∩	#**AC**-∩	Reduction in %
SCOP	298,643	63	99.979
WIKIPEDIA	8,520,420	144	99.998
BIOSQL	5,548,587	276	99.995
LOD	2,396,618	2380	99.901
ENSEMBL	20,911,021	11,473	99.645
CATH	179,532	379	99.789
H-GENOME	78,995,844	6,063,207	92.325
MB	181,535,297	56,241,512	69.019
PDB	223,101,296	9,892,803	95.566
PLISTA	173,045,034	593	99.999

Table 2.5 Characteristics of clusters

D	$\|\mathbf{AC}\|$	$\min \|\mathbf{C}_i\|$	$\max \|\mathbf{C}_i\|$	$\text{median}\|\mathbf{C}_i\|$	$\Sigma\|\mathbf{C}_i\|/\|\mathbf{AC}\|$	*SD*
SCOP	23	1	6	5.0	3.7	2.12
WIKIPEDIA	55	1	6	3.0	2.87	1.35
BIOSQL	144	1	14	2.0	3.1	2.6
LOD	492	1	19	5.0	5.17	3.14
ENSEMBL	2173	1	40	5.0	5.34	2.37
CATH	72	1	11	6.0	5.61	3.32
H-GENOME	287,847	1	135	23.0	21.07	10.07
MB	1,404,892	1	362	39.0	40.03	15.23
PDB	212,305	1	413	33.0	46.61	46.99
PLISTA	194	1	38	3.0	3.78	3.99

2.5.3 Evaluation of the partitioning strategy

To evaluate S-INDD++'s partitioning strategy introduced in Definition 2.3, we compared it with the strategy of BINDER. We conducted the comparison in terms of the number of attributes that each of both algorithms discarded after processing each partition of its Strategy. As in Papenbrock et al. [2015], we set the number of partitions for BINDER to 10 for all experiments, meaning that the size of each BINDER's partition is about 10 % of the corresponding dataset. Again, this number was chosen by Papenbrock et al. [2015] to be the default number for BINDER in all experiments conducted in Papenbrock et al. [2015]. For S-INDD++, we used the partitioning function presented in Example 2.3, which means that S-INDD++'s partitioning consists of four disjoin subsets, where the size of the first one is bout one percent of corresponding dataset size, while the size of the second, the size of the third, and the size of the fourth subset are 10 %, 20 %, and 69 % of the dataset size, respectively.

The results of the evaluation are presented in Tables 2.6 and 2.7, and in Figures 2.5 and 2.6. Table 2.6 presents the percentage of the discarded attributes by S-INDD++ after processing each of the first three partitions, while Table 2.7 presents the percentage of the discarded attributes by BINDER after processing each of the first ninth partitions. Figure 2.5 shows the percentages of survived (active) attributes per S-INDD++'s partition except of the last one, while the percentages of survived attributes per BINDER's partition are shown in Figure 2.6.

S-INDD++ reduces the number of attributes after processing the first partition of each dataset. For instance, the reduction in the number of attributes is 15.8 % (about 61 attributes) for H-GENOME, about 12.2 % (128 attributes) for MB, about 7.4 % (132 attributes) for PDB, and about 12.2 % (17 attributes) for PLISTA after processing the first partition, meaning that S-INDD++ is able to reduce the number of attributes for each dataset after processing only one percent of it. In contrast, BINDER is not able to reduce the number of attributes after processing the first partition of each dataset, except of the first two smallest datasets (SCOP and WIKIPEDIA), although the size of the first BINDER's partition is about 10 % of the total size of the corresponding dataset.

With S-INDD++'s partitioning strategy, the number of discarded attributes continues to increase after processing each partition of each dataset, while with BINDER's strategy, there is no discarded attribute for any dataset after processing the second, the third, and the fourth partition, respectively. Thus, after handling about 40 % of each dataset, except for WIKIPEDIA and SCOP, BINDER is not able to prune any attribute. Furthermore, BINDER's partitioning strategy is useless for MB and PDB because all their attributes were active (involved) in the last partition (the tenth partition). In contrast, S-INDD++ decreases MB's attributes by 17.4% (about 183 attributes) and PDB's attributes by 15.1 % (about 239 attributes) before handling the last partition of each of both datasets. For the other two large datasets H-GENOME and PLISTA, BINDER is able to discard attributes from PLISTA only after the seventh partition, and from H-GENOME only after the ninth partition, while S-INDD++ is able to discard attributes from each of both datasets already after the first partition.

Table 2.6 The percentage of the pruned attributes after processing each of the partitions $\mathbf{P}_i$ $(1 \leq i < 4)$ of S-INDD++'s strategy

D	$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$
SCOP	18.2	22.7	22.7
WIKIPEDIA	78.6	78.6	78.6
BIOSQL	20.8	23.4	26.0
LOD	14.0	17.7	24.4
ENSEMBL	22.3	27.0	33.8
CATH	24.0	32.0	36.0
H-GENOME	15.8	20.7	26.1
MB	12.2	16.1	17.4
PDB	7.4	11.2	13.4
PLISTA	12.2	15.7	17.9

Table 2.7 The percentage of the pruned attributes after processing each of the partitions $\mathbf{P}_i$ $(1 \leq i < 10)$ of BINDER's strategy

D	$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{P}_4$	$\mathbf{P}_5$	$\mathbf{P}_6$	$\mathbf{P}_7$	$\mathbf{P}_8$	$\mathbf{P}_9$
SCOP	22.7	22.7	22.7	22.7	22.7	22.7	22.7	22.7	22.7
WIKIPEDIA	71.4	78.6	78.6	78.6	78.6	78.6	78.6	78.6	78.6
BIOSQL	0	0	0	0	0	0	26	26	26
LOD	0	0	0	0	1.2	1.2	1.2	1.2	1.2
ENSEMBL	0	0	0	0	0	0	0	33.9	34.6
CATH	0	0	0	36	36	36	36	36	36
H-GENOME	0	0	0	0	0	0	0	24.8	25.1
MB	0	0	0	0	0	0	0	0	0
PDB	0	0	0	0	0	0	0	0	0
PLISTA	0	0	0	0	0	0	10	10	10

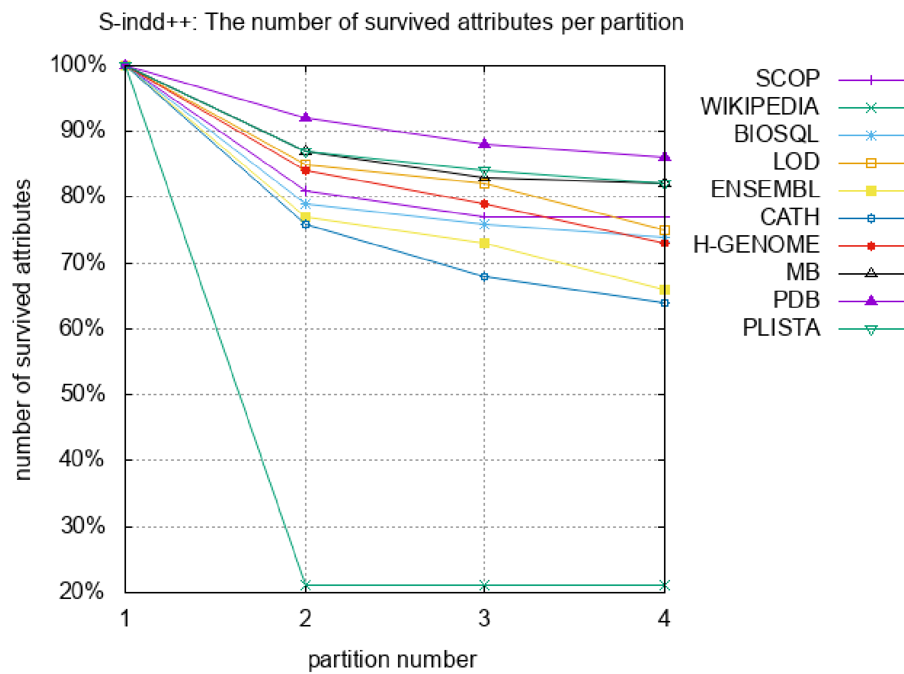


Figure 2.5 The number of active attributes per S-INDD++'s partition for the datasets described in Table 2.2 and Table 2.3

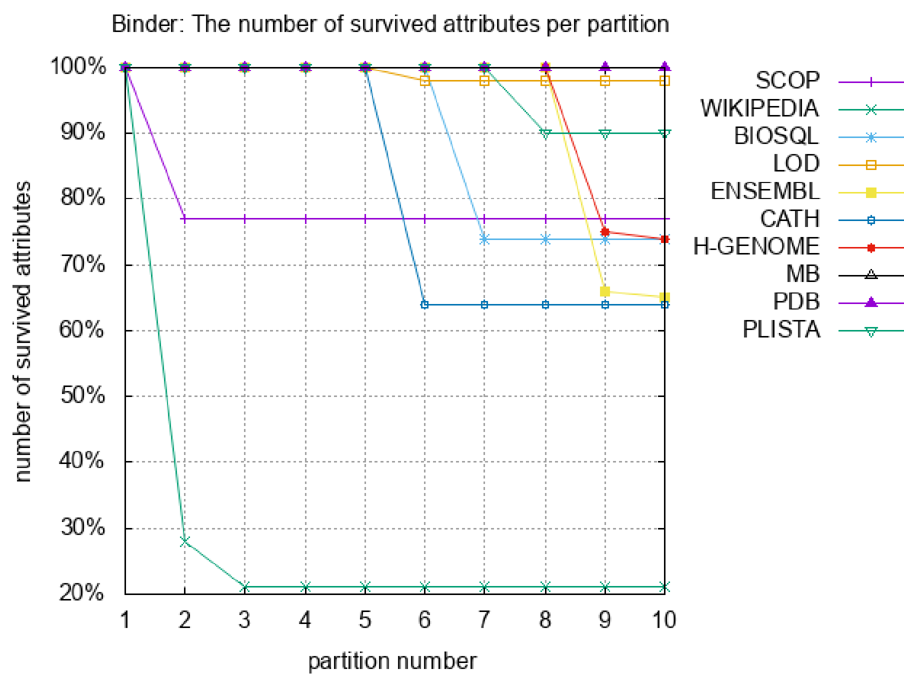
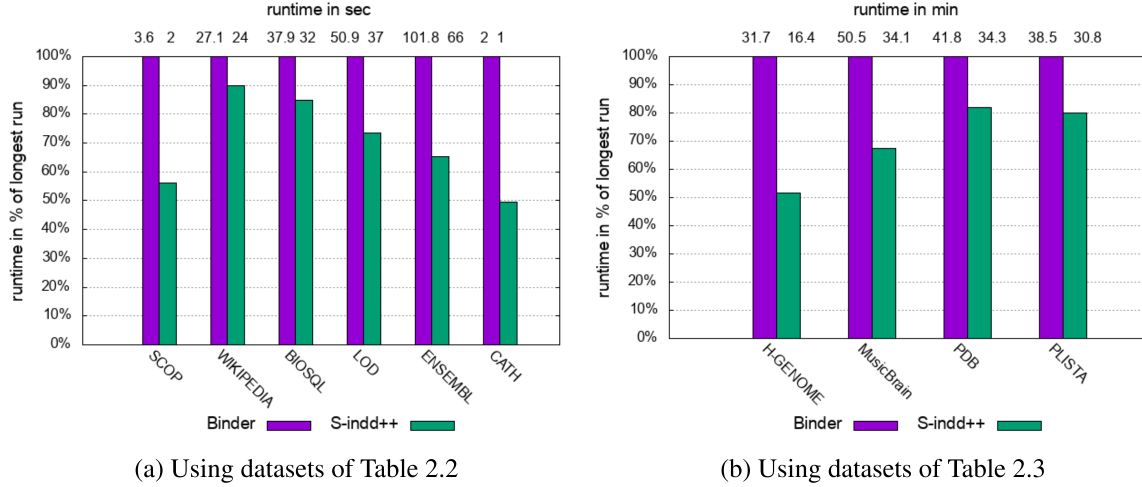


Figure 2.6 The number of active attributes per BINDER's partition for the datasets described in Table 2.2 and Table 2.3



(a) Using datasets of Table 2.2 (b) Using datasets of Table 2.3

Figure 2.7 Comparing S-INDD++'s runtime with BINDER's runtime

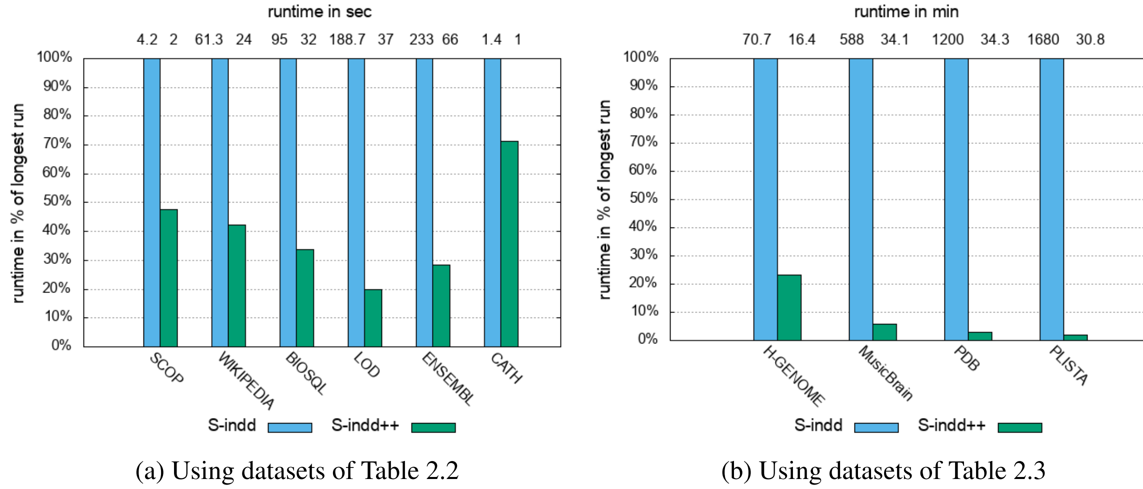
2.5.4 Evaluation of the performance

To evaluate the performance of S-INDD++, we compared its runtime with that of BINDER and with that of S-INDD, respectively.

Comparing with BINDER [Papenbrock et al., 2015] Figure 2.7 compares the runtime of S-INDD++ with that of BINDER for all datasets presented in Table 2.2 and in Table 2.2. It is worth mentioning that all these datasets, except of H-GENOME and MB, were used in Papenbrock et al. [2015] to compare the performance of BINDER with that of SPIDER Bauckmann et al. [2006]. In Figure 2.7, there is a group of two bars for each dataset. In each group, the left bar presents the runtime of BINDER, while the second bar expresses S-INDD++'s runtime as a percentage in the corresponding BINDER's runtime.

For all datasets, S-INDD++ significantly outperforms BINDER. In the group of datasets of the size greater than 11 GB (see Figure 2.7b), S-INDD++ reduces BINDER's runtime by at least 18 % for PDB dataset and to up to 49 % for H-GENOME dataset. In the other group (see Figure 2.7a), the reduction of BINDER's runtime by S-INDD++ ranges from 10 % for BIOSQL to 51 % for CATH dataset. S-INDD++ is falser than BINDER because of the following reasons:

In contrast to S-INDD++, binder needs to redivide each partition that can not fit into main memory, which causes many costly additional I/O-operations. For instance, we observed that BINDER redivided each partition of each dataset in Figure 2.7b into at least three additional partitions, which resulted in a total of at least 30 partitions for each dataset. The way in which S-INDD++ overcomes the need for redividing a partition consists of three stages: i) during the partitioning, S-INDD++ sorts each subset $\mathbf{V}_{j,A}$ of the set $\mathbf{V}_{i,A}$ in the main memory before writing it to the repository of the partition $\mathbf{P}_i$ (see Lines 19-20 in Algorithm 3), ii) if an attribute is not discarded in a partition $\mathbf{P}_k$, S-INDD++ continues sorting the set $\mathbf{V}_{i,A}$ during the processing of the subsequent partition $\mathbf{P}_i$ with $k < i \leq n$



(a) Using datasets of Table 2.2

(b) Using datasets of Table 2.3

Figure 2.8 Comparing S-INDD++'s runtime with S-INDD's runtime

by calling Algorithm 4, which does not need to load all sets $\mathbf{V}_{j,A} \subseteq \mathbf{V}_{i,A}$ $(1 \leq j \leq m)$, which are contained in the repository of $\mathbf{P}_i$, into the main memory at once, as explained in Subsection 2.4.3, and iii) after the termination of Algorithm 4 during the processing of $\mathbf{P}_i$, S-INDD++ generates the attribute clustering $\mathbf{AC}_i$ from all lists L_A of attributes $A \in \mathbf{A}$, which have not been discarded, by calling S-INDD, which does also not need to load the lists L_A into the main memory at once, regardless how many they are and how large they are. The second reason for the better performance of S-INDD++ is that S-INDD++ computes the uINDs of each partition from the clusters, eliminating all redundant intersections caused by computing uINDs from the inverted index, which BINDER has to build for each partition to compute the uINDs of the corresponding partition. The third reason for S-INDD++ to be faster than BINDER is the effectiveness of its partitioning strategy in comparison with BINDER's partitioning strategy, as discussed in details in Subsection 2.5.3.

BINDER's average time is around 38 seconds for the datasets in size less than one GB and around 2438 seconds for the datasets in size greater than 11 GB, while S-INDD++'s average time is around 27 seconds for the datasets in the first group and around 1734 seconds for the datasets in the second group. That means that both algorithms are in average around 64 times slower for the above-11 GB datasets. The reason for these order-of-magnitude changes is that neither BINDER needs to apply repartitioning nor S-INDD++ needs to merge the lists for any dataset in size less than one GB since each relation in these datasets fits into the main memory.

Comparing with S-INDD [Shaabani and Meinel, 2015] Figure 2.8 shows S-INDD++'s runtime compared with that of S-INDD. In this figure, there is a group of two bars for each dataset. In each group, the left bar presents the runtime of S-INDD, while the runtime of S-INDD++ is expressed by the second bar as the percentage in S-INDD's runtime.

As the case for BINDER, S-INDD++ significantly outperforms S-INDD for all datasets. For the datasets in Figure 2.8b, S-INDD++ decreased the runtime of S-INDD by at least 77 % for the case of

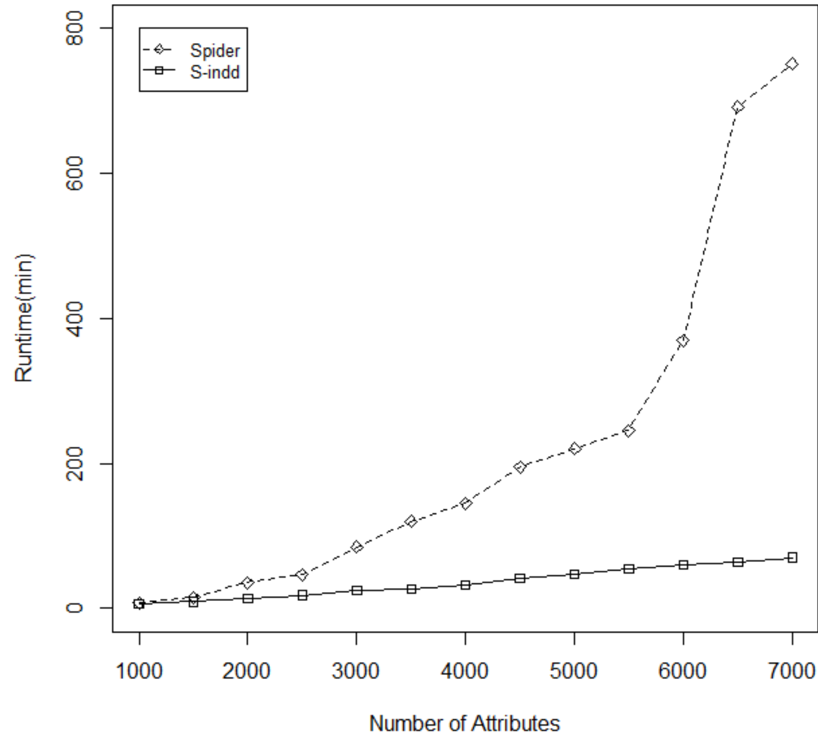


Figure 2.9 Comparing scalability of S-INDD with that of SPIDER by scaling the number of attributes and fixing the number of rows

H-GENOME dataset and by up to 98 % for the case PLISTA. We can even ignore S-INDD++'s runtime needed for PLISTA and PDB in comparison to the runtime needed by S-INDD for both datasets. For the datasets in the other group (see Figure 2.8a), the reduction of the runtime is in the range from 27 % for CATH to 80 % for LOD.

The reason for S-INDD++ being considerably faster than S-INDD is the elimination of sorting the most part of useless attributes. For instance, the total number of discarded attributes before processing the last partition are 101 (26.1 %) attributes for H-GENOME, 183 attributes (17.4 %) for MB, 239 attributes (13.4 %) for PDB, and 25 attributes (17.9 %) for PLISTA, meaning that for the remaining 69% of each of these datasets, S-INDD++ does not need to continue sorting them. These value sets belong to the largest columns with text content whose sorting takes a lot of time. This fact explains why the above-11 GB datasets are harder for S-INDD than for BINDER and for S-INDD++.

It is worth mentioning that the performance of S-INDD depends on the tool used for sorting the attribute values. In these experiments, we sorted the attributes values using our implementation of the external merge-sort algorithms. Another tool is the DBMS, where the performance of sorting is also influenced by its configuration.

2.5.5 Evaluation of the scalability in the number of attributes

The significant difference between S-INDD and SPIDER [Bauckmann et al., 2006] is that SPIDER can only process the whole set of attribute files at once, which makes SPIDER's scalability in the number of attributes decreases when the number of attributes significantly increases. In contrast, S-INDD is a composite of configurable iterations. In each iteration, S-INDD controls the number of attribute files to be processed, which means that we can let S-INDD process all attributes at once by setting $k = |\mathbf{A}| + 1$ in Algorithm 5. Thus, SPIDER is a special case of S-INDD.

To show how significantly S-INDD improves the SPIDER's scalability in the number of attributes, we generated thirteen synthetic datasets with the same number of rows, namely 200,000 rows, and increasing number of attributes. Starting with 1000 different attributes and ten unary INDs in the first dataset, the attributes set in the next dataset consists of the attributes set in the previous dataset plus 500 new different attributes and ten new different INDs so that the thirteenth dataset has 7000 different attributes and 130 unary INDs. For all these datasets , S-INND is configured to aggregate 200 lists ($k = 200$) simultaneously.

Figure 2.9 shows the results of these experiments. (i) For every dataset S-INDD is faster than SPIDER. For example, for the dataset with 7000 attributes and 36.2 GB size, S-INDD needs one hour and ten minutes while SPIDER needs 12 hours and 30 minutes. This means, S-INDD is about 11x faster than SPIDER. (ii) By increasing the number of attributes, SPIDER's runtime grows much faster than S-INDD's runtime. For example, by increasing the number of attributes from 6000 to 7000, SPIDER'runtime increases by 38 % while S-INDD's runtime increases by one percent (for the dataset with 6000 attributes and 31 GB SPIDER needs six hours and 10 minutes while S-INDD needs only about one hour).

2.6 Related work

Bell and Brockhausen [1995] introduced a SQL-based algorithm for detecting uINDs. They generate all uIND candidates from previously collected statistics, such as min-max values and data types. Then, they validate those by using SQL-statements. The transitivity of uINDs is exploited to reduce the number of untested candidates. SQL-based validation, however, is very costly, as shown in Bauckmann [2013].

DeMarchi et al. [2002, 2009] proposed an algorithm that has built the basis for later algorithms for detecting uINDs. It transforms the dataset into an inverted index, in which each value points to the set of all attributes containing that value. Next, the set of all uINDs is computed by intersecting all of those sets, thus causing a lot of redundant intersections as an attribute set in the inverted index can be pointed by a lot of values. Additionally, building the inverted index for the entire dataset at once does not enable the discarding of those attributes that can not be part of any uIND. Moreover, the inverted index does not fit into the main memory for most real-word datasets.

SPIDER proposed by [Bauckmann, 2013; Bauckmann et al., 2006] writes the value set of every attribute to a file after sorting them. Then, SPIDER opens **all files** at once and starts comparing the values in the same way in which merge-sort algorithm does. The result of each comparison is a value pointing to the set of all attributes containing that value. Then, SPIDER uses that set for pruning the set of uIND candidates, similar to DeMarchi's algorithm [DeMarchi et al., 2002, 2009]. SPIDER has solved the problem of not fitting the inverted index into the main memory. SPIDER, however, has two drawbacks: (i) its scalability depends on the number of attributes; and (ii) it causes the same number of redundant intersection operations caused by DeMarchi's algorithm.

S-INDD developed by Shaabani and Meinel [2015] scales well with datasets having a large number of attributes. S-INDD introduces the concept of attribute clustering to eliminates the number of redundant intersections resulting from direct computation of uINDs from the inverted index applied in Bauckmann et al. [2006]; DeMarchi et al. [2002, 2009]; Papenbrock et al. [2015]. Moreover, S-INDD has solved the scalability problem of SPIDER, thus revealing itself as a special case of S-INDD. The drawback of S-INDD, however, is the requirement of sorting the value sets of all attributes before being able to compute the clusters. S-INDD++ [Shaabani and Meinel, 2018b] has solved this issue.

Papenbrock et al. [2015] presented BINDER. In contrast to S-INDD++, BINDER requires the partition to fit into the main memory. It re-divides each partition that does not fit into the main memory, causing costly additional I/O-operations. For each partition, BINDER builds the inverted index used by Bauckmann et al. [2006]; DeMarchi et al. [2002, 2009] to generate the uINDs in the corresponding partition, thereby causing a lot of redundant intersections, which are avoided by S-INDD++, since S-INDD++ relies on S-INDD to compute the uINDs for each partition. The main goal of dividing the dataset is to prune the attributes that can not be part of any uIND. BINDER, however, does not achieve that goal successfully because, for example, BINDER was not able to discard any attribute from some datasets, while S-INDD++ was able to discard attributes from each dataset by processing only 1 % of the dataset (see Tables 2.6, and 2.7; and Figures 2.5, and 2.6 in Subsection 2.5.3).

By applying string concatenations and the Apriori strategy applied by MIND [DeMarchi et al., 2002, 2009], each of S-INDD++; BINDER; S-INDD; and SPIDER can be used for discovering n-ary INDs. But the use of these algorithms for detecting n-ary INDs results in an exponential number of I/O-operations and exponentially increases the original data size. In fact, this idea has been applied by BINDER in Papenbrock et al. [2015] and by SPIDER in Bauckmann [2013], but the success, however, was significantly limited mainly because of the exponential growth of data resulting from value concatenations needed for the application of the principle of candidate generations.

All the approaches previously presented handle the exhaustive search of all valid uINDs. For the approximate discovery of INDs, Dasu et al. [2002] computed a summary of the database using minhash sketches to find potential associations between columns. Thus, INDs can be found efficiently, but with an error: Some discovered INDs are not really satisfied, but also some satisfied INDs can be missed. Recently, Kruse et al. [2017] have presented an approximate discovery of INDs which finds a superset of valid INDs containing false positives (invalid INDs).

DeMarchi et al. [2009] proposed an error measure for the partial satisfaction of an IND. Next, they formulated an algorithm for discovering partial uINDs; this algorithm is similar to DeMarchi's algorithm for exact detection of uINDs. They also discussed the possibility of discovering partial n-ary INDs based on the principle of candidate generation. To provide the user with a condensed set of valid INDs, DeMarchi and Petit [2005] proposed an approach for approximating the set of partial INDs. The basic idea is to relax the user-specified threshold of the error if it can lead to a more condensed set of INDs.

Lopes et al. [2002] used the query-workload-based approach to discover foreign key relationships. From that workload, they derived the set of foreign keys by assuming that join operations are usually performed by equating a key with a foreign key.

Rostin et al. [2009] introduced a machine-learning approach to classify uINDs as primary/foreign keys. They first discovered all uINDs of a given dataset and let each judged by a binary classifier that has been trained on the basis of the features of foreign keys.

Zhang et al. [2010] proposed an approximation algorithm for discovering single-column and multi-column foreign keys. The main idea of their approach is that the value set of the foreign key is a random sample of the value set of the primary key. Therefore, they developed a method to estimate whether the values of foreign key and those of primary key have the same underlying distribution. Before applying that method, they (i) assumed the existence of the primary keys in the dataset and (ii) relaxed the inclusion dependencies that the foreign keys must satisfy. The approximation of the inclusion property is achieved on the basis of the inclusion coefficients, which are estimated by computing a bottom-k sketch for each column in the dataset.

Memari et al. [2015] proposed an approach for profiling single-column and multi-column foreign keys under the different semantics for null markers of the SQL standard. Their approach is based on the techniques proposed by Zhang et al. [2010].

The approaches for the approximate discovery of foreign keys may produce unsatisfied references and may miss satisfied references. For that reason, they focus on the evaluation of precision and recall rather than on the evaluation of the runtime. The specialization on foreign key discovery may also make these approaches inapplicable to other IND use cases, such as data integration [Miller et al., 2001; Schmitt and Saake, 2005], schema redesign [Levene and Vincent, 2000], query optimization [Gryz, 1998], or integrity checking [Casanova et al., 1988].

2.7 Conclusion and future work

We introduced the attribute clustering, a new concept, to derive all valid unary inclusion dependencies more efficiently than to derive them from the inverted index applied by Bauckmann et al. [2006]; DeMarchi et al. [2002, 2009]; Papenbrock et al. [2015].

We then devised S-INDD for the generation of the clusters from large datasets. S-INDD is a composite of configurable computing iterations, in each of which S-INDD controls the number of attributes that have to be processed. This flexibility makes S-INDD scale well in the number of

attributes. In fact, S-INDD has solved the scalability problem of SPIDER [Bauckmann et al., 2006], thereby revealing itself as a special case of S-INDD.

We developed S-INDD++ to achieve two goals: (i) to reduce the time needed by S-INDD for sorting the values of useless attributes–this those attributes that are not part of any valid IND and (ii) to improve the effectiveness of the partition strategy that BINDER [Papenbrock et al., 2015] applies to avoid sorting the attribute value sets. For these proposes, we introduced a new configurable partitioning strategy that enables the pruning of large number of attributes in early phases of the discovery process. Moreover, we extended the attribute clustering to allow S-INDD++ to discard attributes based on the clusters of partitions. Another key feature of S-INDD++ is that S-INDD++ does not require fitting any partition into the main memory, while BINDER needs to re-divide each partition that does not fit into the main memory. The success of S-INDD++ manifested in reducing the sorting time needed by S-INDD by up to 98 % and in reducing BINDER's runtime by up 50 %.

One research question for the future work is how to make S-INDD adaptively select the attributes in each of its iterations. further works should implement a distributed version of S-INDD++ to find whether there can be a significant improvement in the discovery time.

Chapter 3

Incrementally Updating Inclusion Dependencies

3.1 Problem statement

Most datasets are non-static and therefore subject to update. Updates, which are inserts, changes, and deletes, cause metadata to quickly become out-of-date [Agrawal et al., 2012; Fan, 2008; Naumann, 2013; Saha and Srivastava, 2014; Smith et al., 2014].

Example 3.1. *Consider the dataset presented in Table 2.1. The set of valid uINDs in that dataset is* $\mathbf{I} = \{A \subseteq B, D \subseteq B\}$*. Now assume the following two cases:*

(1) Delete: The tuple (b,b,e,a) *(i.e., the second tuple) is deleted from the dataset. This makes* $A \subseteq B$ *invalid because deleting* (b,b,e,a) *makes* $b \in A$ *and* $b \notin B$*. Thus, this deletion changes the current set of valid uINDs* $\mathbf{I}$ *to the set* $\mathbf{I}_1 = \{D \subseteq B\}$*.*

(2) Insert: The new tuple (c,c,e,e) *is inserted into the dataset. This creates a new valid IND, namely* $B \subseteq A$*. Furthermore, it makes* $D \subseteq B$ *invalid. Thus, inserting the tuple* (c,c,e,e) *changes the set* $\mathbf{I}$ *to the set* $\mathbf{I}_2 = \{A \subseteq B, B \subseteq A\}$*.*

The previous example demonstrates that a data change causes new INDs to appear or existing INDs to disappear. It means that the set of uINDs may change and therefore need to be updated after a change in the corresponding dataset.

The current solution to keep the set of uINDs up-to-date after the arrival or deletion of data is to completely rediscover it. This rediscovering process requires the application of one of the existing algorithms to the entire dataset because none of them is suitable for working on dynamic datasets. This solution, however, hurts performance significantly because (i) an initial dataset size is typically bigger than the size of a change in the dataset by several orders of magnitude; and (ii) the performance of the uIND discovering algorithms depends not only on the number of attributes but also on the number of tuples for the most part. Furthermore, rediscovering the set of uINDs in the entire dataset does not take advantage of previously discovered uINDs, which might become only partly invalid after a data

update. Therefore, an incremental approach is the most efficient way to keep the uINDs up-to-date after the arrival or deletion of data [Abedjan et al., 2015; Naumann, 2013; Saha and Srivastava, 2014]. Moreover, the incremental update of uINDs has useful applications, as suggested in Section 3.2.

Consequently, the question that an incremental approach should answer is how to efficiently update the set **I** (i.e., the set of valid uINDs in the dataset) within a short time without processing the entire dataset **D**? Hence, the time needed by an incremental approach for updating the set **I** should be negligible in comparison with the time needed by a static approach. We refer to any algorithm by Bauckmann et al. [2006]; DeMarchi et al. [2009]; Papenbrock et al. [2015]; Shaabani and Meinel [2015, 2018b] for discovering the set **I** as a static discovery of **I**. To define the requirement for the incremental discovery precisely, we denote the insertion of a tuple t into any instance $r \in \mathbf{D}$ by $\mathbf{D} + \{t\}$, and the deletion of t from r by $\mathbf{D} - \{t\}$. We refer to the insertion or deletion of a tuple as $\mathbf{D} \pm \{t\}$.

Definition 3.1. *(Requirement for the incremental discovery) Let $T_{inc}(\{t\})$ be the time needed by an incremental approach for updating* **I** *after the insertion or the deletion of t. Let $T_{st}(\mathbf{D} \pm \{t\})$ be the time needed by a static approach for the discovery of* **I** *in $\mathbf{D} \pm \{t\}$. Then,*

$$T_{inc}(\{t\}) + T_{st}(\mathbf{D} \pm \{t\}) \approx T_{st}(\mathbf{D} \pm \{t\})$$

must be fulfilled.

This definition was inspired by Gruenheid et al. [2014].

Contributions In this chapter, we present the first approach for incrementally updating the set of uINDs when new tuples are inserted and when existing tuples are either deleted or changed (i.e., values are modified). This approach is developed in Shaabani and Meinel [2017, 2018a]. In particular, we made the following contributions:

- We realized the incremental update of uINDs through the incremental update of the attribute clustering. For that purpose, we defined new attribute clustering operations to be applied after each data update.
- We developed algorithms and data structures that efficiently implement the incremental update of the clusters. We also designed cache strategies to reduce the accesses to the external data structures of the algorithms.
- Moreover, we showed how to initialize the data structures for starting the incremental discovery from a non-empty dataset.
- Based on the designed data structures, we demonstrated how to incrementally update approximate uINDs.
- Furthermore, we suggested a sharing-nothing architecture for scaling out the incremental discovery of uINDs.

- We presented the results of exhaustive experiments conducted on five large datasets with hundreds of attributes and more than 116 million tuples.
  - The experiments showed that the incremental approach reduces the runtime of the static discovery by up to 99.9996 %.
  - We evaluated the average time needed for updating the clusters after an insertion or a deletion. In dependence of the dataset, this time varies from 40 to 100 millisecond for an insertion and from 60 to 180 millisecond for a deletion.
  - The evaluation of the cache strategies in terms of the reduction in the accesses of the external data structures showed that in dependence of the dataset, the gained reduction varies from 73 % to 99.99 % for the insertions and from 81 % to 99.99 % for the deletions.
  - Moreover, we evaluated the change of the incremental runtime T_{inc} in correlation with the growth in the number of tuples and the number of attributes, respectively.

The reminder of this chapter is structured as follows: Section 3.2 discusses use cases for incrementally updating uINDs. An overview of the workflow of the system is given in Section 3.3. To achieve the incremental update of the attribute clustering, two operators–the merge operator and the extract operator–are designed in Section 3.4. Section 3.5 presents the developed data structures and the algorithms that efficiently implement the updating process. The incremental discovery of approximate uINDs is discussed in Section 3.6. A distributed architecture for deploying the algorithms and their data structures in a large computation cluster is presented in Section 3.7. Section 3.8 presents the results of a comprehensive evaluation of the algorithms. Section 3.9 presents the related work, while the last section concludes and discusses future works.

3.2 Use cases for incrementally updating INDs

3.2.1 Query optimization

One strategy for query optimization is query rewriting, which tries to reformulate parts of queries with semantically equivalent and more efficient query terms [Gryz, 1998]. Let $R[A_1,\ldots,A_{|R|}]$ and $S[B_1,\ldots,B_{|S|}]$ be two relations for which the unary IND $A_i \subseteq B_j$ (for an $i \in \{1,\ldots,|R|\}$ and a $j \in \{1,\ldots,|S|\}$) has been defined as a foreign key relationship at the time of the modeling (designing) of R and S. Then, consider the SQL query presented in Figure 3.1.

The query joins the relation R with the relation S. If we know that $B_j \subseteq A_i$ holds; which leads to $A_i = B_j$ because we already have $A_i \subseteq B_j$, then the join is superfluous and can be removed from the query [Gryz, 1998]. This means that every time the previous query has to be executed, we infer from the attribute clustering whether $B_j \subseteq A_i$ is valid. If it is valid, then we remove the join from the query.

```
SELECT R.A_1, ..., R.A_|R|
FROM R, S
WHERE R.A_i = S.B_j
AND R.A_1 = 'a';
```

Figure 3.1 Example for SQL-query that can be optimized every time when $B_j \subseteq A_i$ becomes valid after incrementally updating uINDs

3.2.2 Schema update and data linkage

Let **D** be a large legacy dataset over a set of relations **R**. We assume that there are business requirements for extending **R** to include a new set of relations **S** where initially the dataset **G** over **S** is empty. In this real scenario, it is necessary to connect **D** and **G** by finding joining paths and foreign-key relationships between them. Since inclusion dependencies can indicate such relationships, they are indispensable for this task. Expert users, however, often fail to specify the INDs between the attributes of **R** and the attributes of **S** for various reasons–for instance, they have a difficult time identifying INDs when **D** contains hundreds of tables, thousands of attributes, and insufficient (or missing) documentation [Zhang et al., 2010].

Applying the incremental discovery helps to reveal the uINDs between the relation instances in $\mathbf{D} \cup \mathbf{G}$ over the time. Instead of completely rediscovering the uINDs in $\mathbf{D} \cup \mathbf{G}$ by applying the static discovery every time the user might think that $\mathbf{D} \cup \mathbf{G}$ contains enough data to identify the uINDs, uINDs can be dynamically updated and periodically presented to the user in a way that can help to identify the join paths and the foreign-keys. For instance, If i) we associate a counter $changeCount(A \subseteq B)$ with every uIND $A \subseteq B$ and initialize it with zero, and ii) we increase $changeCount(A \subseteq B)$ by one every time $A \subseteq B$ changes from invalid to valid or from valid to invalid, then for each valid IND $A \subseteq B$, the lower its counter, the higher the likelihood of $A \subseteq B$ being a foreign-key. That can be attributed to the fact that the counter of a valid IND indicates the number of violences of its validity. The basis of this approach is that if uINDs are known at the designing time of the database, then database designers define them as foreign-key relationships and expect them not to be violated during the update of the database. Of course, this approach is only put forward as a suggestion as it requires further investigation and evaluation.

3.2.3 Data integration

Many organizations have adopted the Microservices Architecture to solve the problems of monolithic applications. Instead of building a single monstrous application, the idea of Microservices is to split the application into a set of smaller interconnected services. A service typically implements a set of distinct features or functionality, such as order management, customer management [Newman, 2015].

The Microservices Architecture significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema–this often results in duplication of some data. Thus, in this architecture,

matching the database schemata of the services and transforming their data into a joined representation are crucial tasks for having a data model across the enterprise; in fact, such a model is inevitable for maximizing the value of the data for analysis. [Renz et al., 2016]. Since schema-matching methods rely heavily on structural metadata, such as INDs [Miller et al., 2001], updating the databases of services can be accompanied by incrementally updating the uINDs between them to incrementally compare their respective data. This comparison can help to determine which attributes contain overlapping or identical value sets and which are redundant and can be eliminated. Moreover, with the help of uINDs, we can identify attributes containing the same data but with different names (synonyms) and those which have the same name but different semantic (homonyms) [Evoke Software, 2000; Schmitt and Saake, 2005]. We can also apply the approach presented in Subsection 3.2.2 for the incrementally identifying the join paths between the datasets.

Furthermore, if some uINDs between datasets of the application have been semantically specified at the modeling time, then the validity of these uINDs can be monitored by the incremental update. Every time after updating all uINDs, we check whether the specified uINDs are still valid. If some of them are invalid, then the user will be notified. This kind of monitoring can help to insure the quality of data.

3.3 Workflow overview

Figure 3.2 gives an overview of the workflow of our system for incrementally detecting uINDs. The system enqueues every tuple immediately after its insertion into the dataset or its deletion from it, and also checks the queue periodically. If the queue is not empty, a tuple will be dequeued to be processed. In the first step, the processing of a tuple consists of mapping each value v in the tuple to the set of all tuple attributes to which this value belongs. We denote such a set of tuple attributes as $\Delta\mathbf{C}$. This mapping is a special case of attribute clustering introduced in Section 2.2 of Chapter 2. For instance, after dequeuing the tuple $t = (c, c, e, e)$, it is mapped to the set $\{(c, \{A, B\}), (e, \{C, D\})\}$.

Then, for each $(v, \Delta\mathbf{C})$, the system calls Algorithm 8 to handle the insertion if the dequeued tuple has been inserted. Otherwise, it calls Algorithm 13 to handle the deletion. Both algorithms are presented in Section 3.5. Algorithms 8 and 13 work on a set of data structures presented in Subsection 3.5.1 to keep the attribute clustering of the entire dataset up-to-date after the insertion or deletion of a tuple.

Keeping the attribute clustering up-to-date is one of the key points in our design because all valid uINDs are efficiently derivable from the attribute clustering, as explained in Section 2.2 of Chapter 2.

Updating the attribute clustering $\mathbf{AC}$ of a dataset $\mathbf{D}$ is based on two operators applied on $\mathbf{AC}$ and the attribute clustering $\Delta\mathbf{AC}$ of the dequeued tuple referred to as t. If t is inserted, then we apply the merge operator (see Definition 3.2). Otherwise, we apply the extract operator (see Definition 3.4). The result of the merge operator is the attribute clustering of $\mathbf{D} + \{t\}$, while the result of the extract operator is the attribute clustering of $\mathbf{D} - \{t\}$.

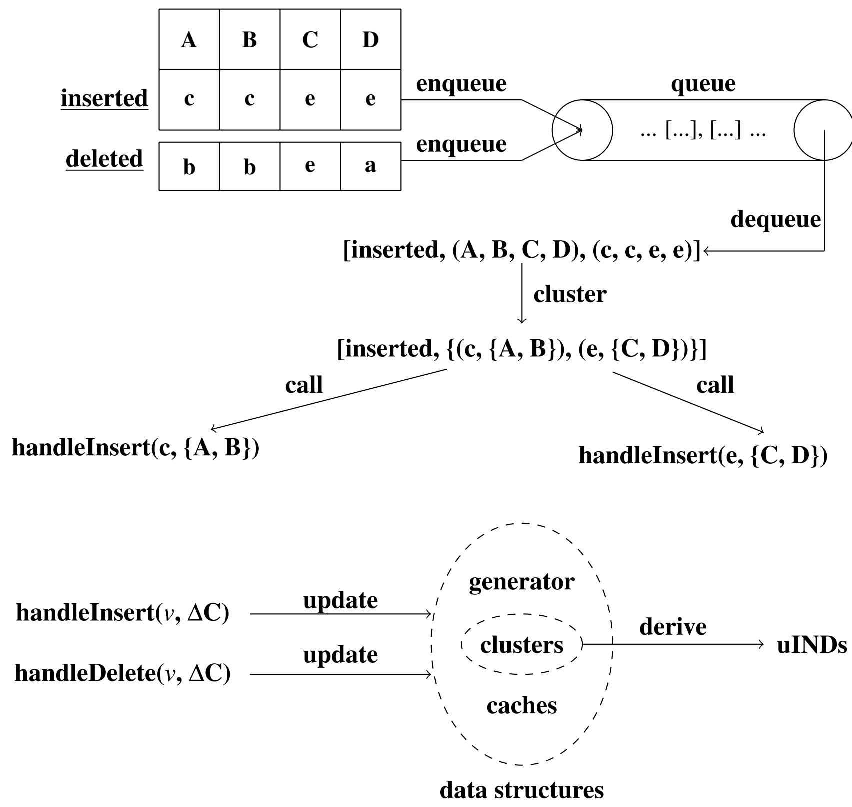


Figure 3.2 Workflow overview

After a modification of an existing tuple, the attribute clustering can be updated by a composite operation consisting of the merge operation and the extract operation, as illustrated in Section 3.4.

The queue will be almost empty if the time between two consecutive dataset operations is less than the time T_{inc} needed by the system to update the attribute clustering after the data change (i.e., in this case, updating the attribute clustering will be in real time). The specific implementation of the queue can vary from a simple queue in the main memory to a dedicated queuing server. In our implementation, the queue is based on the file system. In section 3.7, we extend this architecture for deploying the system in a large computation cluster.

3.4 Attribute clustering operations

Our purpose is to update the attribute clustering $\mathbf{AC}$ of $\mathbf{D}$ after inserting or deleting a tuple t, meaning that our goal is two-fold: (i) to compute the attribute clustering of $\mathbf{D}+\{t\}$ from $\mathbf{AC}$ and $\{t\}$ after inserting t into $\mathbf{D}$, and (ii) to compute the attribute clustering of $\mathbf{D}-\{t\}$, also from $\mathbf{AC}$ and $\{t\}$, but after deleting t from $\mathbf{D}$. To achieve these goals, we compute the attribute clustering of $\{t\}$, referred to as $\Delta\mathbf{AC}$, then (i) we define the merge operator on $\mathbf{AC}$ and $\Delta\mathbf{AC}$ to obtain the attribute clustering of $\mathbf{D}+\{t\}$, and (ii) we define the extract operator on $\mathbf{AC}$ and $\Delta\mathbf{AC}$ to obtain the attribute clustering of the dataset $\mathbf{D}-\{t\}$.

Attribute clustering of a tuple Let t be a one-tuple instance of a relation $R \in \mathbf{R}$. We denote the set of all values occurring in t by $\Delta\mathbf{V}$. Accordingly, the attribute clustering over $\Delta\mathbf{V}$ is $\Delta\mathbf{AC}$ and the generator of $\Delta\mathbf{AC}$ is Δf.

Example 3.2. *We consider Figure 3.2. For* $\{t\} = \{(c,c,e,e)\}$, *we have* $\Delta\mathbf{V} = \{c,e\}$, $\Delta f(c) = \{A,B\}$, *and* $\Delta f(e) = \{C,D\}$. *Thus,* $\Delta\mathbf{AC} = \{\{A,B\},\{C,D\}\}$.

3.4.1 Merge operator

The merge operator merges the attribute clustering $\mathbf{AC}$ of $\mathbf{D}$ and the attribute clustering $\Delta\mathbf{AC}$ of an inserted tuple t to produce the *attribute clustering* $\mathbf{AC}+\Delta\mathbf{AC}$ of $\mathbf{D}+\{t\}$.

Definition 3.2. *(Merge operator) Let* $f : \mathbf{V} \to 2^{\mathbf{A}}$ *be the generator of* $\mathbf{AC}$, *and* $\Delta f : \Delta\mathbf{V} \to 2^{\mathbf{A}}$ *be the generator of* $\Delta\mathbf{AC}$. *We define* $f+\Delta f : \mathbf{V}\cup\Delta\mathbf{V} \to 2^{\mathbf{A}}$ *as*

$$(f+\Delta f)(v) = \begin{cases} f(v)\cup\Delta f(v) & \text{if } v \in \mathbf{V}\cap\Delta\mathbf{V} \\ f(v) & \text{if } v \in \mathbf{V}\setminus\Delta\mathbf{V} \\ \Delta f(v) & \text{if } v \in \Delta\mathbf{V}\setminus\mathbf{V} \end{cases}$$

The merging of $\mathbf{AC}$ *and* $\Delta\mathbf{AC}$ *is* $\mathbf{AC}+\Delta\mathbf{AC} = (f+\Delta f)(\mathbf{V}\cup\Delta\mathbf{V})$.

Lemma 3.1. $\mathbf{AC}+\Delta\mathbf{AC}$ *is the attribute clustering over* $\mathbf{V}\cup\Delta\mathbf{V}$ *and* $f+\Delta f$ *is its generator.*

Proof. For any $v \in \mathbf{V} \cup \Delta\mathbf{V}$, we have to show that $(f + \Delta f)(v)$ is the maximum set of attributes whose value sets contain v.

i) For $v \in \mathbf{V} \cap \Delta\mathbf{V}$, we assume that $(f + \Delta f)(v) = f(v) \cup \Delta f(v)$ is not the maximum set of attributes whose value sets contain v. This means there is some attribute A with $v \in \mathbf{V} \cap \Delta\mathbf{V}$, $A \notin f(v)$ and $A \notin \Delta f(v)$. $v \in \mathbf{V}_A$ and $A \notin f(v)$ contradict the fact that $f(v) \in \mathbf{AC}$. $v \in \Delta\mathbf{V}_A$ and $A \notin \Delta f(v)$ contradict the fact that $\Delta f(v)$ is a cluster in $\Delta\mathbf{AC}$. Thus, our assumption is wrong.

ii) The case $v \in \mathbf{V} \setminus \Delta\mathbf{V}$ means that inserting t does not add v to the column of any attribute $A \in \mathbf{A}$, meaning that v has the same cluster in $\mathbf{D}$ and in $\mathbf{D} + \{t\}$. Thus, we have $(f + \Delta f)(v) = f(v)$.

iii) The case $v \in \Delta\mathbf{V} \setminus \mathbf{V}$ means that v does not occur in $\mathbf{D}$ and, after inserting t, occurs in $\mathbf{D} + \{t\}$ in the columns of all attributes in $\Delta f(v)$. Thus, $\Delta f(v)$ is the cluster of v in $\mathbf{D} + \{t\}$. □

Example 3.3. *What is the attribute clustering of the dataset in Table 2.1 after inserting (c, c, e, e)? According to Figure 2.1 and Example 3.2 and since $c, e \in \mathbf{V} \cap \Delta\mathbf{V}$, we have*

$$(f + \Delta f)(c) = \{B, D\} \cup \{A, B\} = \{A, B, D\}$$
$$(f + \Delta f)(e) = \{C\} \cup \{C, D\} = \{D, C\}$$

For $x \in \{a, b, d, f\} \subset \mathbf{V} \setminus \Delta\mathbf{V}$, we have

$$(f + \Delta f)(x) = f(x)$$

Thus, $\mathbf{AC} + \Delta\mathbf{AC} = \{\{A, B, D\}, \{A, B\}, \{C, D\}, \{C\}\}$. Based on Lemma 2.3 in Chapter 2, the set of unary INDs is now $\mathbf{I} = \{A \subseteq B, B \subseteq A\}$.

3.4.2 Extract operator

The extract operator extracts the attribute clustering $\Delta\mathbf{AC}$ of a deleted tuple t from the attribute clustering $\mathbf{AC}$ of $\mathbf{D}$ to produce the attribute clustering $\mathbf{AC} - \Delta\mathbf{AC}$ of $\mathbf{D} - \{t\}$. But before defining the extract operator, we have to identify the set of distinct values in $\mathbf{D} - \{t\}$. For this purpose, and for every $v \in \mathbf{V}$ and every attribute $A \in \mathbf{A}$, we define the variable F_A^v indicating the frequency of occurrence of the value v in the column of A before deleting t. Then, we introduce the following operator.

Definition 3.3. *(Bag difference) We call the set*

$$\mathbf{V} \setminus_b \Delta\mathbf{V} = \mathbf{V} \setminus \{v \in \Delta\mathbf{V} \mid (\forall A \notin \Delta f(v) : F_A^v = 0) \wedge (\forall A \in \Delta f(v) : F_A^v = 1)\}$$

the bag difference between $\mathbf{V}$ and $\Delta\mathbf{V}$.

Now, it is obvious that the set of values in $\mathbf{D} - \{t\}$ is $\mathbf{V} \setminus_b \Delta\mathbf{V}$.

Definition 3.4. *(Extract operator) Let* $f : \mathbf{V} \to 2^{\mathbf{A}}$ *be the generator of* $\mathbf{AC}$*, and* $\Delta f : \Delta\mathbf{V} \to 2^{\mathbf{A}}$ *be the generator of* $\Delta\mathbf{AC}$*. We define* $f - \Delta f : \mathbf{V} \setminus_b \Delta\mathbf{V} \to 2^{\mathbf{A}}$ *as*

$$(f - \Delta f)(v) = \begin{cases} f(v) & \text{if } v \notin \Delta\mathbf{V} \\ f(v) \setminus \{A \in \Delta f(v) \mid F_A^v = 1\} & \text{if } v \in \Delta\mathbf{V} \end{cases}$$

The extract of $\Delta\mathbf{AC}$ *from* $\mathbf{AC}$ *is* $\mathbf{AC} - \Delta\mathbf{AC} = (f - \Delta f)(\mathbf{V} \setminus_b \Delta\mathbf{V})$.

Lemma 3.2. $\mathbf{AC} - \Delta\mathbf{AC}$ *is the attribute clustering over* $\mathbf{V} \setminus_b \Delta\mathbf{V}$ *and* $f - \Delta f$ *is its generator.*

Proof. For any $v \in \mathbf{V} \setminus_b \Delta\mathbf{V}$, we have to show that $(f - \Delta f)(v)$ is the maximum set of attributes whose value sets contains v.

i) For the case $v \notin \Delta\mathbf{V}$, there is no change in the cluster of v in $\mathbf{D}$.
Thus, $(f - \Delta f)(v) = f(v)$ is the cluster of v in $\mathbf{D} - \{t\}$.

ii) For $v \in \Delta\mathbf{V}$, $(f - \Delta f)(v)$ is the cluster $f(v)$ after removing each attribute $A \in \Delta f(v)$ whose column does not contain v anymore after deleting t. Thus, after deleting t from $\mathbf{D}$, $(f - \Delta f)(v)$ becomes the cluster of v in $\mathbf{D} - \{t\}$. □

Example 3.4. *What is the attribute clustering of the dataset in Table 2.1 after deleting* (b, b, e, a)*? For this tuple, we have*

$$\Delta\mathbf{V} = \{b, e, a\}$$
$$\Delta f(b) = \{A, B\}, \Delta f(e) = \{C\}, \text{ and } \Delta f(a) = \{D\}$$

For $v = c, d$ *or* $f \notin \Delta\mathbf{V}$*, we have*

$$(f - \Delta f)(v) = f(v)$$

For $v = b$*, we have*

$$(f - \Delta f)(b) = \{A, B\} \setminus \{B\} = \{A\}$$

because $F_A^b = 2$ *and* $F_B^b = 1$ *(i.e., we do not remove* $A \in \Delta f(b)$ *from* $f(b) = \{A, B\}$ *because* $F_A^b > 1$*, but we remove* $B \in \Delta f(b)$ *from* $f(b)$ *because* $F_B^b = 1$*).*
For $v = e$*, we have* $F_C^e = 1$*. Therefore,*

$$(f - \Delta f)(e) = \{C\} \setminus \{C\} = \emptyset$$

For $v = a$*, we have* $F_D^a = 2 > 1$*. Therefore,*

$$(f - \Delta f)(a) = f(a) = \{A, B, D\}$$

Thus, $\mathbf{AC} - \Delta\mathbf{AC} = \{\{A, B, D\}, \{A, B\}, \{B, D\}, \{A\}, \{C\}\}$*. Based on Lemma 2.3 in Chapter 2, the set of unary INDs is now* $\mathbf{I} = \{D \subseteq B\}$*.*

Updating AC after a change of an existing tuple The following example shows how to update the attribute clustering after a change of an existing tuple.

Example 3.5. *Assume that the tuple (b,c,f,c) (i.e., the third tuple in Table 2.1) has to be modified to be (b,g,f,g), meaning that (i) the value c has to be deleted once from the column of attribute B and once from the column of attribute D, and (ii) the value g has to be added once to the column of B and once to the column of D. We consider deleting c from B and D as deleting the tuple $t_1 = (B = c, D = c)$ from* **D***, while we consider adding g to B and D as inserting the tuple $t_2 = (B = g, D = g)$ into* **D**. *The attribute clustering of t_1 is $\Delta\mathbf{AC}_1 = \{B,D\}$ over $\Delta\mathbf{V}_1 = \{c\}$ with the generator $\Delta f_1(c) = \{B,D\}$, while the attribute clustering of t_2 is $\Delta\mathbf{AC}_2 = \{B,D\}$ over $\Delta\mathbf{V}_2 = \{g\}$ with the generator $\Delta f_2(g) = \{B,D\}$. Thus, updating* **AC** *after changing the tuple (b,c,f,c) consists of the following two operations: (i) extracting $\Delta\mathbf{AC}_1$ from* **AC**, *and (ii) merging* **AC** *with $\Delta\mathbf{AC}_2$.*

3.5 Algorithms

In this section, we design the data structures to organize and access the clusters and the corresponding generator. Then, we develop the algorithms for the manipulation of these data structures according to the attribute clustering operations. After analyzing the performance of the algorithms, we present an approach to initialize the data structures for the case in which the incremental discovery starts with a non-empty dataset.

3.5.1 Data structures

Data structure of the clusters For every cluster $\mathbf{C} \in \mathbf{AC}$, we define a record $r_{\mathbf{C}} = (cid, \mathbf{C})$ in which $cid \in \mathbb{N}$ identifies the cluster **C** uniquely in **AC**. We denote the set of all these records $r_{\mathbf{C}}$ by *ACRecs*. To efficiently support the retrieval and deletion of a cluster by giving its identifier, we define an index on *ACRecs*. The keys of this index are the identifiers and the entries are the clusters. Furthermore, we define a second hash index to efficiently retrieve an identifier by giving the corresponding cluster. The keys in the second index are the clusters and the entries are the identifiers.

Data structure of the generator For every value $v \in \mathbf{V}$, we define a record $r_v = (v, F_1^v, \ldots, F_{|\mathbf{A}|}^v), cid)$, in which *cid* is the identifier of the cluster to which v belongs and F_i^v $(1 \leq i \leq |\mathbf{A}|)$ is the frequency of v occurrence in the column of A_i. We denote the set of all these records r_v by *GenRecs*. Every record $r_v \in$ *GenRecs* is uniquely identified by the corresponding value v because a value can belong to only one cluster (see Definition 2.1). We implement *GenRecs* as a relational table, which means that all operations defined on *GenRecs* are SQL queries. To efficiently retrieve, update, and delete a value record by giving the value, we define an index on the values. We also define an index on the identifiers of the clusters to efficiently retrieve and count the values belonging to a certain cluster.

We implement the generator in terms of an external data structure because of the number of values in **V**. This number can be so large that *GenRecs* does not fit into the main memory, which is also

Table 3.1 Data structures for the generator and the clusters of the dataset in Table 2.1

v	F_A^v	F_B^v	F_C^v	F_D^v	cid
a	1	1	0	2	2
b	2	1	0	0	1
c	0	1	0	2	3
d	1	1	0	0	1
e	0	0	1	0	4
f	0	0	3	0	4

cid	cluster
2	$\{A,B,D\}$
1	$\{A,B\}$
3	$\{B,D\}$
4	$\{C\}$

why we separate clusters from the data structure of the generator, meaning that we keep and manage the data structure of the clusters in the main memory. Table 3.1 presents the data structures of the generator for the dataset in Table 2.1 and the corresponding clusters in Figure 2.1.

Data structures for accessing the generator While updating *GenRecs* and *ACRecs*, a cluster $\mathbf{C}$ will be removed from *ACRecs* if the number of values belonging to it is zero. This requires that each time we have to know whether a cluster $\mathbf{C}$ has to be removed, we have to access *GenRecs* to compute the number of values belonging to the cluster. To reduce the number of required SQL queries for this case, we define a record $(cid, vcount)$ for every cluster. The variable *vcount* is a counter of the values that have been added to, retrieved from, and/or deleted from *GenRecs* for the cluster identified by cid. We denote the set of all such records by *CountCache*. Thus, if $vcount \geq 1$ for a cluster $\mathbf{C}$, we can immediately decide that $\mathbf{C}$ is to not be removed from *ACRecs*. Otherwise, if there is no record in *CountCache* for $\mathbf{C}$ or if there is record with $vcount = 0$, we have to retrieve the number of values belonging to $\mathbf{C}$ from *GenRecs*. To efficiently update and delete a counter in *CountCache*, we define an index with the cluster identifiers as keys and the counters as entries.

For every value $v \in \Delta\mathbf{V}$, we have to determine whether it is new (i.e., $v \notin \mathbf{V}$). If v is not new (i.e., $v \in \mathbf{V} \cap \Delta\mathbf{V}$), we have to know its cluster, which requires querying *GenRecs*. To reduce the database accesses in this case, we cache the record (v, cid) for every value that has been added to or retrieved from *GenRecs*. We refer to the set of all such records as *ValCache*. Thus, if there is a record in *ValCache* for an input value v, we immediately know that v is not new. Otherwise, we have to query *GenRecs*. If the result of the query is empty, we conclude that there is no cluster to which v belongs and, therefore, v is new. To efficiently retrieve a record from *ValCache*, we define an index with the values as keys and the cluster identifiers as entries. We refer to *ValCache* and *CountCache* in the rest of the paper as *cache strategies*.

3.5.2 Handling insertion

Algorithm 8 implements the merge operator introduced in Definition 3.2. It receives a value v and the set $\Delta\mathbf{C}$ that contains all attributes into whose columns v has been inserted. If the value is new (i.e.,

Algorithm 8: handleInsert

```
  Input  : v, ΔC
1 cid ← getClusterID(v)
2 if cid = null then
3 |   handleInsertingNewValue(v, ΔC)
4 else
5 |   handleInsertingExistingValue(v, ΔC, cid)
```

Algorithm 9: getClusterID

```
  Input  : v
  Output : cid
1 cid ← ValCache·getCID(v)
2 if cid = null then
3 |   cid ← GenRecs·getCID(v)
4 |   if cid ≠ null then
5 |   |   ValCache·add(v, cid)
6 |   |   CountCache·init(cid, 1)
```

Algorithm 10: handleInsertingNewValue

```
  Input  : v, ΔC
1 cid ← ACRecs·getCID(ΔC)
2 if cid = null then
3 |   cid ← ACRecs·add(ΔC))
4 if cid ∈ CountCache then
5 |   CountCache·increase(cid))
6 else
7 |   CountCache·init(cid, 1)
8 ValCache·add(v, cid)
9 GenRecs·add(v, cid, ΔC)
```

Algorithm 11: handleInsertingExistingValue

```
   Input   : v, ΔC, cid
1  C ← ACRecs·getCluster(cid)
2  C′ ← C ∪ ΔC
3  if C′ = C then
4  |  GenRecs·iUpdate(v, ΔC)
5  else
6  |  cid′ ← ACRecs·getCID(C′)
7  |  if cid′ = null then
8  |  |  cid′ ← ACRecs·add(C′)
9  |  if cid′ ∈ CountCache then
10 |  |  CountCache·increase(cid′)
11 |  else
12 |  |  CountCache·init(cid′, 1)
13 |  GenRecs·iUpdate(v, cid′, ΔC)
14 |  ValCache·update (v, cid′)
15 |  CountCache·decrease(cid)
16 |  handleDeletingCluster(cid)
```

$v \notin \mathbf{V}$), it does not belong to any cluster in *ACRecs*. Consequently, there is no record for v in *GenRecs*. Otherwise, v already exists in the dataset and belongs to a cluster. Algorithm 8 calls Algorithm 9 to identify whether the value v is new.

Algorithm 9 returns the NULL-marker if the value v does not exist either in the cache *ValCache* or in the set *GenRecs*. Otherwise, it returns the identifier of the cluster to which v belongs. In the beginning, Algorithm 9 tries to find the identifier of the cluster of v in the cache *ValCache* (Line 1). If the cache *ValCache* does not contain a record (v, cid) for the value v (Line 2), Algorithm 9 queries *GenRecs* (Line 3). If it has found a record for the value v in *GenRecs* (Line 4), it adds v and the identifier of its cluster to the cache *ValCache* (Line 5) and creates a value counter for the cluster and initializes it with one, i.e., it adds the record $(cid, 1)$ to *CountCache* (Line 6).

Handling the insert of a new value Algorithm 10 implements the merge operation for the case $v \in \Delta\mathbf{V} \setminus \mathbf{V}$. In this case, we have $(f + \Delta f)(v) = \Delta f(v) = \Delta\mathbf{C}$ according to Definition 3.2 and Lemma 3.1, meaning that the attribute set $\Delta\mathbf{C}$ is the cluster of the new value v. But $\Delta\mathbf{C}$ is not necessarily a new cluster because $\Delta\mathbf{C}$ can already exist in the set *ACRecs* if there is some value v' in *GenRecs* that is different from v and has $\Delta\mathbf{C}$ as a cluster. To identify whether $\Delta\mathbf{C}$ already exists in *ACRecs*, Algorithm 10 retrieves *ACRecs* for the existing of $\Delta\mathbf{C}$ (Line 1). The operation *getCID(*$\Delta\mathbf{C}$*)* on *ACRecs* returns the identifier of $\Delta\mathbf{C}$ if $\Delta\mathbf{C}$ exists in *ACRecs*; otherwise, it returns the NULL-marker. If $\Delta\mathbf{C}$ does not exist in *ACRecs*, Algorithm 10 adds $\Delta\mathbf{C}$ to *ACRecs* (Lines 2-3). Now and after knowing the identifier of $\Delta\mathbf{C}$, Algorithm 10 has to update *GenRecs*, *ValCache*, and *CountCache*, increasing the number of values belonging to $\Delta\mathbf{C}$ by one if the identifier of $\Delta\mathbf{C}$ exists in *CountCache*. Otherwise,

it creates a value counter for $\Delta\mathbf{C}$ and initializes it with one (Line 7). Notice that the value counter for a cluster exits in *CountCache* only if a value belonging to the cluster has been handled. For *ValCache*, Algorithm 10 adds the new value v with the identifier of its cluster to *ValCache* (Line 8), and adds a new record $(v, F_1^v, \ldots, F_{|\mathbf{A}|}^v, cid)$ to *GenRecs* (Line 9) in which F_i^v $(1 \leq i \leq |\mathbf{A}|)$ is initialized as follows:

$$F_i^v = \begin{cases} 1 & \text{if } A_i \in \Delta\mathbf{C} \\ 0 & \text{if } A_i \notin \Delta\mathbf{C} \end{cases} \tag{3.1}$$

We initialize F_i^v with one for each $A_i \in \Delta\mathbf{C}$ because v is new and has been added only once to each column of $A_i \in \Delta\mathbf{C}$.

Handling the insert of an existing value Algorithm 11 implements the merge operator for the case $v \in \mathbf{V} \cap \Delta\mathbf{V}$. Based on Definition 3.2 and Lemma 3.1, the new cluster of the value v is $\mathbf{C}' = (f + \Delta f)(v) = f(v) \cup \Delta f(v) = \mathbf{C} \cup \Delta\mathbf{C}$, in which $\mathbf{C}$ is the current v's cluster identified by cid. If $\Delta\mathbf{C} \subseteq \mathbf{C}$, we have $\mathbf{C}' = \mathbf{C}$. For this case, there is no change in *ACRecs*. But what Algorithm 11 needs to do here is only to update the v's record $(v, F_1^v, \ldots, F_{|\mathbf{A}|}^v, cid)$ in *GenRecs* as follows:

$$F_i^v = \begin{cases} F_i^v + 1 & \text{if } A_i \in \Delta\mathbf{C} \\ F_i^v & \text{if } A_i \notin \Delta\mathbf{C} \end{cases} \tag{3.2}$$

We increase F_i^v for each $A_i \in \Delta\mathbf{C}$ by one because v has been added to the column of each $A_i \in \Delta\mathbf{C}$.

If $\mathbf{C}' \neq \mathbf{C}$, Algorithm 11 has to find out i) whether or not $\mathbf{C}'$ already exists in *ACRecs*, and ii) whether or not $\mathbf{C}$ has to be deleted from *ACRecs*. If $\mathbf{C}'$ does not exist in *ACRecs*, then Algorithm 11 adds $\mathbf{C}'$ to *ACRecs* (Lines 7-8). After that, Algorithm 11 has to take care of *CountCache*. If there is no value count for $\mathbf{C}'$, Algorithm 11 creates a new value count for $\mathbf{C}'$ and initializes it with one. In the other case, it increases the value count by one. Notice that a value count for a cluster exists in *CountCache* only if a value belonging to this cluster has been inserted before.

Since the cluster of v has become $\mathbf{C}'$, the records of v in *GenRecs* and in *ValCache* have to be updated. The updating of v's record $r_v = (v, F_1^v, \ldots, F_{|\mathbf{A}|}^v, cid)$ consists of replacing the identifier cid by the identifier cid' of the new cluster of v and of applying the Formula 3.2. The updating of the record (v, cid) in *ValCache* consists of only replacing cid by cid' (Lines 13-14).

Now, the value v does not belong to $\mathbf{C}$ anymore. Therefore, Algorithm 11 decreases the number of values belonging to the cluster $\mathbf{C}$ by one (Line 15) and calls Algorithm 12, which has to decide whether $\mathbf{C}$ has to be deleted.

Deleting a cluster When a value has been deleted or assigned to a different cluster, we have to check whether its previous cluster has to be deleted from *ACRecs*. A cluster has to be deleted if the number of values belonging to it is zero. Algorithm 12 performs this check. If the cluster (i.e., its identifier) does not exist in *CountCache* or the associated number of values in *CountCache* is zero,

Algorithm 12: handleDeletingCluster

```
   Input   : cid
1  vc ← 0
2  if cid ∈ CountCache then
3  |  vc ← CountCache·getCount(cid)
4  |  if vc = 0 then
5  |  |  vc ← GenRecs·getCount(cid)
6  else
7  |  vc ← GenRecs·getCount(cid)
8  if vc = 0 then
9  |  ACRecs·remove(cid)
10 |  if cid ∈ CountCache then
11 |  |  CountCache·remove(cid)
12 else
13 |  if cid ∉ CountCache then
14 |  |  CountCache·init(cid, vc)
```

Algorithm 12 retrieves the number of values from *GenRecs* (Lines 2-7). If this number is zero, the cluster will be removed from *ACRecs*, and also from *CountCache* if it exists in *CountCache* (Lines 8-11).

If the number of values retrieved from *GenRecs* is greater than zero and the cluster does not exist in *CountCache*, Algorithm 12 creates a value count for the cluster in *CountCache* and initializes it with the retrieved number (Lines 12-14). By doing so, Algorithm 12 will reduce the querying *GenRecs* in the future. Accessing and querying *GenRecs* is more expensive than querying *CountCache* because *GenRecs* is an external data structure in the form of a relational table.

3.5.3 Handling deletion

Algorithm 13 implements the extract operator defined in Definition 3.4, handling the deletion of a value v from the columns of the attributes in $\Delta\mathbf{C}$. Notice that $\Delta\mathbf{C}$ can only be a subset of the current cluster $\mathbf{C}$ of v because $\mathbf{C}$ is the maximum set of attributes whose columns contain v (see Definition 2.1). Based on Definition 3.4 and Lemma 3.2, we obtain the new cluster $\mathbf{C}'$ of v from the current cluster $\mathbf{C}$ after removing some attributes in $\Delta\mathbf{C}$ from $\mathbf{C}$ (Lines 1-7). Each attribute $A_i \in \Delta\mathbf{C}$ has to be removed from $\mathbf{C}$ if the value v occurs only once in the column of A_i (i.e., $F_i^v = 1$).

If the deleted value v occurs more than once in the column of each $A_i \in \Delta\mathbf{C}$ (i.e., $\forall A_i \in \Delta\mathbf{C} : F_i^v > 1$), the new cluster $\mathbf{C}'$ is identical to the current cluster $\mathbf{C}$. In this case, Algorithm 13 needs only to update the value record in *GenRecs* as follows:

$$F_i^v = \begin{cases} F_i^v - 1 & \text{if } A_i \in \Delta\mathbf{C} \\ F_i^v & \text{if } A_i \notin \Delta\mathbf{C} \end{cases} \tag{3.3}$$

Algorithm 13: handleDelete

```
    Input   : v, ΔC
 1  cid ← getClusterID(v)
 2  C ← ACRecs·getCluster(cid)
 3  C' ← C
 4  {F_i^v | A_i ∈ ΔC} ← GenRecs·getFreqs(v, ΔC)
 5  for F_i^v ∈ {F_i^v | A_i ∈ ΔC} do
 6      if F_i^v ≤ 1 then
 7          C' ← C' \ {A_i}
 8  if C = C' then
 9      GenRecs·dUpdate(v, ΔC)
10  else
11      if C' = ∅ then
12          GenRecs·remove(v)
13          ValCache·remove(v)
14      else
15          cid' ← ACRecs·getCID(C')
16          if cid' = null then
17              cid' ← ACRecs·add(C')
18              CountCache·init(cid', 1)
19          else
20              CountCache·increase(cid')
21          ValCache·update(v, cid')
22          GenRecs·dUpdate(v, cid', ΔC)
23      CountCache·decrease(cid)
24      handleDeletingCluster(cid)
```

Algorithm 13 decreases F_i^v for each $A_i \in \Delta\mathbf{C}$ by one because v has been deleted only once from the column of each $A_i \in \Delta\mathbf{C}$.

If the new cluster $\mathbf{C}'$ is different from $\mathbf{C}$, the value v does not belong to $\mathbf{C}$ anymore. Therefore, Algorithm 13 reduces the number of values belonging to $\mathbf{C}$ (Line 23), and then calls Algorithm 12 (see Subsection 3.5.2) to decide whether $\mathbf{C}$ has to be deleted from *ACRecs* (Line 24).

The case in which the new cluster $\mathbf{C}'$ is empty occurs if $\mathbf{C} = \Delta\mathbf{C}$ and $F_i^v = 1$ for each $A_i \in \Delta\mathbf{C}$. This means that the value v has been completely deleted from the dataset $\mathbf{D}$. Therefore, Algorithm 13 deletes the value v from *GenRecs* and from *ValCache* (Lines 12-13).

If the new cluster $\mathbf{C}'$ is not empty and different from $\mathbf{C}$, Algorithm 13 has to know whether $\mathbf{C}'$ already exists in *ACRecs*. For the case that $\mathbf{C}'$ does not exist in *ACRecs*, Algorithm 13 adds it to *ACRecs*, creates a new value count for $\mathbf{C}'$ in *CountCache*, and initializes this counter with one (Lines 16-18). For the other case in which $\mathbf{C}'$ already exists, Algorithm 13 only increases the number of values belonging to $\mathbf{C}'$ (Line 20).

The last step Algorithm 13 has to execute for the case $\mathbf{C}' \neq \emptyset$ and $\mathbf{C}' \neq \mathbf{C}$ is to update the records of v in *GenRecs* and in *ValCache* respectively (Lines 21-22). Updating the v's record in *ValCache* consists of only replacing the previous cluster identifier *cid* by *cid*$'$. While Updating the record $r_v = (v, F_1^v, \ldots, F_{|\mathbf{A}|}^v, cid)$ of v consists of decreasing F_i^v for each $A_i \in \Delta\mathbf{C}$ by one (see Formula 3.3), and replacing the previous cluster identifier *cid* by *cid*$'$.

3.5.4 Performance analysis

As the data structure of the generator is external, the performance of our algorithms depends mainly on querying and updating it. Therefore, we analyze the performance of the algorithms in terms of the number of accesses needed for querying and updating *GenRecs* (i.e., the generator).

Identifying the cluster of an input value To find the identifier of the cluster to which an input value v belongs, Algorithm 9 needs one query to retrieve the identifier from *GenRecs* (Line 3) if *ValCache* does not contain a record for v. Otherwise, Algorithm 9 does not need any access to *GenRecs*. Thus, in the worst case, we need one generator access to identify the cluster of a value.

Deleting a cluster Algorithm 12 queries *GenRecs* to identify the number of values belonging to a cluster, whose deletion comes into consideration (Line 5 or 7). This query is required if the value count in *CountCache* is zero or if *CountCache* does not contain an entry for the cluster. Thus, in the worst case, we need one access to *GenRecs* to know whether a cluster has to be deleted.

Handling an insert If the input value v is new, Algorithm 10 needs one generator access for inserting a new value record r_v for v into *GenRecs* (Line 9). If the input value v is not new, Algorithm 11 also needs one generator access to update the value record r_v in *GenRecs* (Line 4 or 13). Thus, in the best

case, Algorithm 8 needs one generator access to update the attribute clustering, while it needs three accesses in the worst case.

Handling a delete Algorithm 13 needs one access to *GenRecs* to compute the frequency of occurrence of the input value in each attribute from whose column the value has been deleted (Line 4). Furthermore, Algorithm 13 needs to access *GenRecs* either to remove the value record r_v from *GenRecs* or to update it (Line 12 or 22). Therefore, for handling the delete of a value, we need two generator accesses to *GenRecs*. Thus, in the best case Algorithm 13 needs two accesses to the external data structure for updating the attribute clustering after deleting a value from the dataset, while it needs four accesses to *GenRecs* in the worst case.

We now formulate the results of the previous analysis as follows:

Lemma 3.3. *Let* $\Delta\mathbf{V}$ *be the set of distinct values occurring in a tuple* t *inserted into* $\mathbf{D}$*. Updating the attribute clustering after inserting* t *needs* $|\Delta\mathbf{V}|$ *generator accesses in the best case, and* $3 \times |\Delta\mathbf{V}|$ *generator accesses in the worst case.*

Lemma 3.4. *Let* $\Delta\mathbf{V}$ *be the set of distinct values occurring in a tuple* t *deleted from* $\mathbf{D}$*. Updating the attribute clustering after deleting* t *needs* $2 \times |\Delta\mathbf{V}|$ *generator accesses in the best case, and* $4 \times |\Delta\mathbf{V}|$ *generator accesses in the worst case.*

This means that in both cases–the best case and the worst–updating the generator after deleting a tuple t needs $|\Delta\mathbf{V}|$ more database accesses than updating it after inserting t, where $\Delta\mathbf{V}$ is the set of distinct values occurring in t. Notice that updating the generator without the cache strategies always requires $3 \times |\Delta\mathbf{V}|$ accesses after an insertion, and $4 \times |\Delta\mathbf{V}|$ after a deletion.

For every input value $v \in \Delta\mathbf{V}$, an access to the generator is either (i) a modification of v's record, (ii) removing v's record from the generator, or (iii) inserting the v's record into the generator. Removing a record from the generator or inserting a record may require more runtime than a modification of a record because the former operation does not cause any reorganization of the index defined on the values, while the latter two operations may cause such reorganization of that index.

Furthermore, the cost of updating the generator depends on two variables: the number of its records and the number of attributes. This means that the performance of updating the generator increases if the number of distinct values grows or the number of attributes grows.

3.5.5 Initializing the data structures

If the incremental discovery starts working on a non-empty dataset, we have to initialize the data structures with the clusters and frequency of occurrences of each value $v \in \mathbf{V}$ in each attribute $A \in \mathbf{A}$ (i.e., F_A^v) computed from such a dataset. To handle this case, we present a new algorithm that compute the clusters and the quantities F_A^v at the same time. The presented algorithm is an extension of S-INDD [Shaabani and Meinel, 2015].

Overall Idea For each value $v \in \mathbf{V}$, we compute the set $\mathbf{F}^v$ that is defined as

$$\mathbf{F}^v = \{(A, F_A^v) \mid A \in \mathbf{A} \wedge v \in \mathbf{V}_A\}$$

Each tuple $(A, F_A^v) \in \mathbf{F}^v$ consists of an attribute A with $v \in \mathbf{V}_A$ and of the frequency of v's occurrence in the column of A. Notice that for each $(A, F_A^v) \in \mathbf{F}^v$, we have $F_A^v > 0$ because of $v \in \mathbf{V}_A$. Then, the cluster of v is the set

$$\mathbf{C} = \{A \mid \exists (A, F_A^v) \in \mathbf{F}^v\}$$

because this set is the maximum set of all attributes whose value sets contain v. Thus, for each value $v \in \mathbf{V}$, we can derive from the set $\mathbf{F}^v$ (i) the cluster of v and (ii) the frequency of occurrence of v in each attribute $A \in \mathbf{A}$.

Computing the sets $\mathbf{F}^v$ To compute the sets $\mathbf{F}^v$ ($v \in \mathbf{V}$), we export the value set $\mathbf{V}_A$ for each attribute $A \in \mathbf{A}$ from the dataset $\mathbf{D}$ and create a list L_A that consists of all elements of the set

$$\{(v, \{(A, F_A^v)\}) \mid v \in \mathbf{V}_A\}$$

and that is sorted according to the values of $\mathbf{V}_A$. We store each of these lists in a file in an external repository $\mathbf{L}$ (hard disk).

Example 3.6. *For the dataset presented in Table 2.1, the repository* $\mathbf{L}$ *initially has the following four lists.*

$$L_1 = [(a, \{(A,1)\}), (b, \{(A,2)\}), (d, \{(A,1)\})]$$
$$L_2 = [(a, \{(B,1)\}), (b, \{(B,1)\}), (c, \{(B,1)\}), (d, \{(B,1)\})]$$
$$L_3 = [(e, \{(C,1)\}), (f, \{(C,3)\})]$$
$$L_4 = [(a, \{(D,2)\}), (c, \{(D,2)\})]$$

Based on these lists, Algorithm 14 generates the sets $\mathbf{F}^v$ iteratively, meaning that in each iteration, a part of some sets of the sets $\mathbf{F}^v$ will be computed. This iterative generation consists of a sequence of merging operations. Each merging operation reads k ($2 \leq k \leq |\mathbf{A}|$) lists $L_1, L_2, \ldots, L_k$ from the repository $\mathbf{L}$ and replaces them with a new list L whose elements are generated as follows (see Algorithms 15 and 16): Each group of tuples $(v, \mathbf{F}_{l_1}^v), (v, \mathbf{F}_{l_2}^v), \ldots, (v, \mathbf{F}_{l_n}^v)$ ($\{l_1, \ldots, l_n\} \subseteq \{1, \ldots, k\}$) that share a value v and that have the property:

$$\forall l_i \in \{l_1, \ldots, l_n\} : (v, \mathbf{F}_{l_i}^v) \in L_{l_i} \wedge \mathbf{F}_{l_i}^v \subseteq \mathbf{F}^v$$

will be selected to generate the tuple $(v, \mathbf{F}_x^v)$ with

$$\mathbf{F}_x^v = \cup_{l_i \in \{l_1, \ldots, l_n\}} \mathbf{F}_{l_i}^v \subseteq \mathbf{F}^v$$

Algorithm 14 repeats the merging of lists (Lines 1-2) until the repository **L** has fewer than k lists. Notice that every new list generated by a merging operation has to be stored as a temporary result in **L** (Line 9 in Algorithm 15). The following example illustrates the emerging operation.

Example 3.7. *Based on Example 3.6 and for $k = 3$, Algorithm 14 has to execute only one merging operation. If the lists L_1, L_2, and L_3 are selected for merging, then the list*

$$L_{1,2,3} = [(a, \{(A,1),(B,1)\}),(b, \{(A,2),(B,1)\}),(c, \{(B,1)\}),\\ (d, \{(A,1),(B,1)\}),(e, \{(C,1)\}),(f, \{(C,3)\})]$$

will be generated and the repository **L** *will be changed to contain only $L_{1,2,3}$ and L_4.*

For an efficient implementation of the merging operation and for managing a simultaneous reading of k lists (files) from the repository **L**, a priority queue is used by Algorithm 15 (and also by Algorithm 16). The queue manages k readers (sequential file readers). Every reader is associated with a list and points to the entry that can currently be read from the list. For every two readers, *Reader1* and *Reader2*, *Reader1* has a higher priority than *Reader2* if and only if the value v in $(v, \mathbf{F}^v)$ is smaller than or equal to the value v' in $(v', \mathbf{F}^{v'})$, with $(v, \mathbf{F}^v)$ being the tuple *Reader1* can currently read and $(v', \mathbf{F}^{v'})$ being the tuple *Reader2* can currently read.

The purpose of using a priority queue is to enable an efficient collection of all sets $\mathbf{F}^v_{l_1}, \ldots, \mathbf{F}^v_{l_n}$ ($\{l_1, \ldots, l_n\} \subseteq \{1, \ldots, k\}$) by a simultaneous and sequential reading of k lists where v is the smallest value among all values that have not been read from the k lists in the queue yet. That is possible in a simultaneous sequential reading because the lists are sorted according to the values $v \in \mathbf{V}$ and the priority in the queue is defined according to the ascending order of the values. This kind of application of the priority queue is well known by external merge-sort algorithms.

The purpose of the parameter k is to circumvent the limitation of the maximum number of files that the underlying operation system can open at the same time.

Computing the clusters and initializing the generator After finishing the merging, Algorithm 14 generates the clusters and initializes the generator by processing all remaining k' ($1 \leq k' < k$) lists simultaneously (Lines 3-10).

For every value v, there are still l_v ($1 \leq l_v < k'$) lists containing tuples of the form $(v, \mathbf{F}^v_i)$ ($1 \leq i \leq l_v$). Collecting all these remaining tuples and computing the union $\cup_{1 \leq i \leq l_v} \mathbf{F}^v_i$ by Algorithm 16 results in finishing the generation of the set $\mathbf{F}^v$ for the value v. After that, Algorithm 14 computes the cluster of v (Line 6). Then, Algorithm 14 adds it to the cluster records *ACRecs* if *ACRecs* does not contain it. After obtaining the identifier of the computed cluster, the value record of v will be created in Line 10 of Algorithm 14 based on the following formula:

$$F^v_i = \begin{cases} F^v_{A_i} & \text{if } (A_i, F^v_{A_i}) \in \mathbf{F}^v \\ 0 & \text{if } (A_i, F^v_{A_i}) \notin \mathbf{F}^v \end{cases} \tag{3.4}$$

```
Algorithm 14: initDS
   Input   : L, A, k
 1 while (L contains k or more than k lists) do
 2     mergeLists(L, k)
 3 Queue ← createPriorityQueue(L)
 4 while Queue·size() ≠ 0 do
 5     (v, F^v) ← collectNextAttSets(Queue)
 6     C ← {A | ∃(A, F_A^v) ∈ F^v}
 7     cid ← ACRecs·getCID(C)
 8     if cid = null then
 9         cid ← ACRecs·add(C)
10     GenRecs·add(v, cid, F^v)
```

```
Algorithm 15: mergeLists
   Input   : L, k
 1 L_1, L_2, ..., L_k ← selectLists(L, k)
 2 Queue ← createPriorityQueue(L_1, L_2, ..., L_k, L)
 3 L ← [ ]
 4 while Queue·size() ≠ 0 do
 5     (v, F^v) ← collectNextAttSets(Queue)
 6     L ← L + [(v, F^v)]
 7 remove(L_1, L2, ..., L_k, L)
 8 write(L, L)
```

```
Algorithm 16: collectNextSets
   Input   : Queue
   Output  : (v, F^v)
 1 F^v ← ∅
 2 repeat
 3     Reader ← Queue·pull()
 4     (v, G^v) ← Reader·readCurrent()
 5     (v, F^v) ← (v, F^v ∪ G^v)
 6     if Reader·hasNext() then
 7         Reader·readNext()
 8         Queue·add(Reader)
 9     Reader' ← Queue·peek()
10     (v', G^v') ← Reader'·readCurrent()
11 until (Queue·size() = 0) ∨ (v' ≠ v)
```

Example 3.8. *For $k = 3$ and based on Example 3.7, the repository* $\mathbf{L}$ *contains the following two lists after merging.*

$$L_{1,2,3} = [(a,\{(A,1),(B,1)\}),(b,\{(A,2),(B,1)\}),(c,\{(B,1)\}),(d,\{(A,1),(B,1),(5,\{C\}),(e,\{(C,1)\}),(f,\{(C,3)\})]$$
$$L_4 = [(a,\{(D,2)\}),(c,\{(D,2)\})]$$

For the value $v = a$, Algorithm 16 creates the set

$$\mathbf{F}^a = \{a,\{(A,1),(B,1),(D,2)\}\}$$

from the tuple

$$(a,\{(A,1),(B,1\}) \in L_{1,2,3}$$

and the tuple

$$(a,\{(D,2)\}) \in L_4.$$

From $\mathbf{F}^a$ *Algorithm 14 computes* $\mathbf{C} = \{A,B,D\}$*, i.e., the cluster of a, and then adds it to ACRecs. After that, Algorithm 14 inserts the record* $(a,1,1,0,2,\underline{2})$ *into GenRecs, where the identifier of* $\mathbf{C}$ *in ACRecs is* $\underline{2}$ *(cf. Table 3.1).*

It is worth mentioning that during the whole merging process, the repository size remains almost constant. That is because (i) the selected k lists in every merging operation will not be needed after they are merged, which allows Algorithm 15 to remove them from the repository after merging them (see Line 8), and (ii) the size of the new list resulting from the merging of the selected k lists can not exceed the total size of these k lists.

3.6 Incrementally updating approximate inclusion dependencies

Approximate INDs are introduced and justified in DeMarchi et al. [2009]. An approach for the detection of approximate unary INDs in static datasets is presented in that work. In this section, we develop a novel algorithm for incrementally discovering approximate unary INDs by utilizing the data structures developed in Subsection 3.5.1.

The definition below is adopted from DeMarchi et al. [2009].

Definition 3.5. *(Approximate uIND) Given an user-specified threshold* $\varepsilon \in]0,1]$*, an approximate unary IND between two attributes* $A,B \in \mathbf{A}$ *is satisfied with respect to* ε*, denoted by* $A \subseteq_\varepsilon B$*, if and only if* $1 - |\mathbf{V}_A \cap \mathbf{V}_B|/|\mathbf{V}_A| \leq \varepsilon$.

We denote the set of all approximate unary IND by $\widetilde{\mathbf{I}}$:

$$\widetilde{\mathbf{I}} = \{A \subseteq_\varepsilon B \mid A,B \in \mathbf{A} \wedge (1 - |\mathbf{V}_A \cap \mathbf{V}_B|/|\mathbf{V}_A|) \leq \varepsilon\}$$

Notice that for an uIND $A \subseteq B \in \mathbf{I}$ we have $1 - |\mathbf{V}_A \cap \mathbf{V}_B|/|\mathbf{V}_A| = 0 < \varepsilon$, because $\mathbf{V}_A \subseteq \mathbf{V}_B$. Thus, we have

$$\mathbf{I} \subseteq \widetilde{\mathbf{I}} \tag{3.5}$$

Now, our goal is to efficiently compute $|\mathbf{V}_A|$ and $|\mathbf{V}_A \cap \mathbf{V}_B|$ from *GenRecs* and *ACRecs* for all $A, B \in \mathbf{A}$ with $A \subseteq B \notin \mathbf{I}$ to decide the satisfaction of $A \subseteq_\varepsilon B$ according to Definition 3.5.

Based on the value frequencies F_A^v, we observe

$$\forall v \in \mathbf{V} : F_A^v > 0 \Leftrightarrow v \in \mathbf{V}_A \tag{3.6}$$

Therefore, we have

$$|\mathbf{V}_A| = |\{v \in \mathbf{V} \mid F_A^v > 0\}| \tag{3.7}$$

Thus, we can compute $|\mathbf{V}_A|$ from the generator by counting each value record with $F_A^v > 0$. But that counting is not efficient because the data structure of the generator is sparse regarding the variables F_A^v, meaning that the percentage of records with $F_A^v = 0$ is higher than that with $F_A^v > 0$. Therefore, we have to reduce the number of records that have to be checked if $F_A^v > 0$.

For that purpose, we define the set CID_A as

$$CID_A = \{cid \in \mathbb{N} \mid \exists (cid, \mathbf{C}) \in ACRecs : A \in \mathbf{C}\}$$

That is, CID_A is the set of all identifiers of clusters containing A. With the help of CID_A, we have the following inclusion:

$$\mathbf{V}_A \subseteq \{v \mid \exists (v, \ldots, F_A^v, \ldots, cid) \in GenRecs : cid \in CID_A\} \tag{3.8}$$

Inclusion 3.8 is correct because: (i) for any cluster with $cid \in CID_A$, there is a value $v \in \mathbf{V}_A$, and possibly, some values $v' \in \mathbf{V} \setminus \mathbf{V}_A$ belonging to that cluster, and (ii) a value in $\mathbf{V}_A$ can not belong to a cluster with an identifier that is not contained in CID_A. From 3.6 and 3.8 we conclude

$$\mathbf{V}_A = \{v \mid \exists (v, \ldots, F_A^v, \ldots, cid) \in GenRecs : cid \in CID_A \wedge F_A^v > 0\} \tag{3.9}$$

From Formula 3.9, we conclude that the number of value records that have to be checked if $F_A^v > 0$ is $|CID_A|$, which is smaller than $|\mathbf{V}|$. Moreover, identifying the value records with cluster identifiers in CIA_A is an efficient operation because the value records in *GenRecs* are indexed by the cluster identifiers (see Subsection 3.5.1). Therefore, counting the values of $\mathbf{V}_A$ based on 3.9 is more efficient than counting them based on 3.7.

Analog to the computation of $|\mathbf{V}_A|$, we compute $|\mathbf{V}_A \cap \mathbf{V}_B|$ based on

$$\mathbf{V}_{AB} = \{v \mid \exists (v, \ldots, cid) \in GenRecs : cid \in CID_{AB} \wedge F_A^v > 0 \wedge F_B^v > 0\} \tag{3.10}$$

Algorithm 17: Computation of approximate unary INDs

```
Input  : I, ε
Output : Ĩ
1  Set2Size ← createMap(Set, Int)
2  Ĩ ← ∅
3  for A ∈ A do
4      I̅_A ← {B | A ⊆ B ∉ I}
5      if I̅_A ≠ ∅ then
6          CID_A ← ACRecs·getClusterIDs(A)
7          |V_A| ← GenRecs·countValues(A, CID_A)
8          for B ∈ I̅_A do
9              if {A, B} ∉ Set2Size then
10                 CID_AB ← ACRecs·getClusterIDs(A, B)
11                 |V_A ∩ V_B| ← GenRecs·countValues (A, B, CID_AB)
12                 Set2Size·put({A, B}, |V_A ∩ V_B|)
13             |V_A ∩ V_B| ← Set2Size·get ({A, B})
14             if 1 − |V_A ∩ V_B|/|V_A| ≤ ε then
15                 Ĩ ← Ĩ ∪ {A ⊆_ε B}
```

where

$$CID_{AB} = \{cid \in \mathbb{N} \mid \exists(cid, \mathbf{C}) \in ACRecs : A, B \in \mathbf{C}\}$$

Algorithm 17 is the formulation of the ideas developed in this section for incrementally discovering approximate unary INDs. According to Formula 3.5, Algorithm 17 takes into account only pairs A, $B \in \mathbf{A}$ such that $A \subseteq B \notin \mathbf{I}$ because such pairs are the candidates to be in $\widetilde{\mathbf{I}} \setminus \mathbf{I}$. Therefore, it computes the set $\overline{\mathbf{I}_A}$ (Line 3). Then, for each $A \in \mathbf{A}$ and $B \in \overline{\mathbf{I}_A}$, Algorithm 17 computes $|\mathbf{V}_A|$ (Lines 6-7) and $|\mathbf{V}_A \cap \mathbf{V}_B|$ (Lines 10-11) according to Formula 3.9 and Formula 3.10 respectively. Algorithm 17 uses hash map to cache $|\mathbf{V}_A \cap \mathbf{V}_B|$ to avoid computing it twice, in the case of which neither $A \subseteq B$ nor $B \subseteq A$ are in $\mathbf{I}$ because, in that case, we have to check the satisfaction of both $A \subseteq_\varepsilon B$ and $B \subseteq_\varepsilon A$. Algorithm 17 computes the set $\widetilde{\mathbf{I}}$ incrementally because both data structures *ACRecs* and *GenRecs* are incrementally updated.

Notice that taking Algorithm 14 for initializing the data structures, Lemma 2.3 for deriving $\mathbf{I}$, and Algorithm 17 together define an approach for discovering approximate unary INDs in a static dataset. That is valuable because the data structures of the algorithm for approximate unary INDs presented by DeMarchi et al. [2009] do not fit into main memory for most real-word datasets.

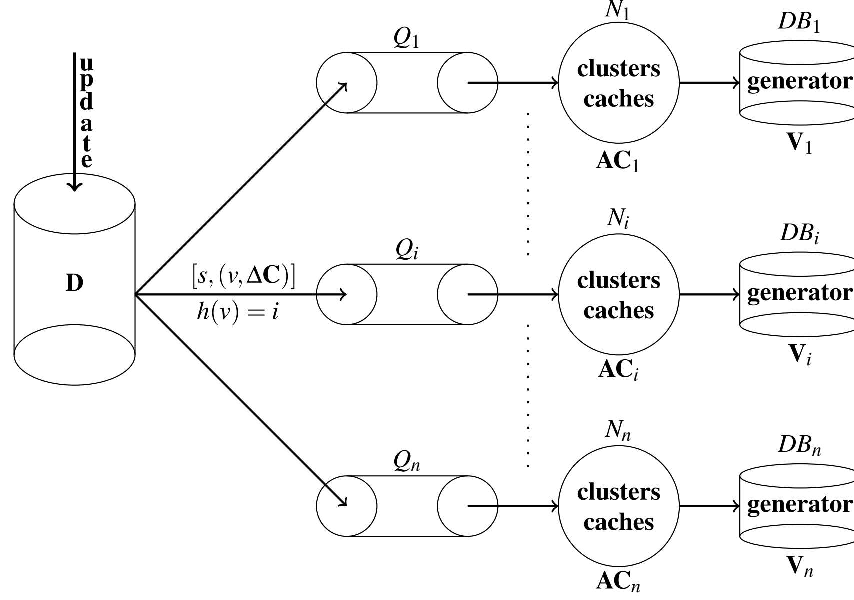


Figure 3.3 Computation cluster for incrementally updating of uINDs based on partitioning the dataset into n disjoint partitions. The partition number of a value $v \in \mathbf{V}$ is calculated via hash-partitioning h.

3.7 Scaling out the incremental discovery of INDs

Now, we suggest a distributed shared-nothing architecture for deploying the proposed incremental discovery of uINDs in a large computation cluster. Figure 3.3 shows the design of this architecture, which is based on partitioning the dataset into $n \geq 1$ disjoint subsets.

Describing the design The architecture shown in Figure 3.3 is based on the partitioning $\mathbf{P} = \{\mathbf{P}_i \mid 1 \leq i \leq n\}$. Each partition $\mathbf{P}_i$ is assigned to the massaging queue Q_i, the computation node N_i, and the database DB_i. The node N_i $(1 \leq i \leq n)$ is responsible for incrementally updating the attribute clustering $\mathbf{AC}_i$ over the value set $\mathbf{V}_i$. It manages the data structure *ACRecs* and the caches for the clusters in $\mathbf{AC}_i$ and the values in $\mathbf{V_i}$. The database DB_i is responsible for storing, updating, and querying the generator of $\mathbf{AC}_i$. Hence, each partition has its own data structures: its own cluster records *ACRecs*, its own generator records *GenRecs* and its own caches–this means that for each two nodes N_i and N_j with $i \neq j$, updating the attribute clustering $\mathbf{AC_i}$ is independent of updating the attribute clustering $\mathbf{AC}_j$.

The Queue Q_i $(1 \leq i \leq n)$ is assigned to a node N_i and delivers it with the events $[s,(v,\Delta\mathbf{C})]$, where i) $h(v) = i$, ii) $\Delta\mathbf{C}$ is the cluster of v in $\Delta\mathbf{AC}$ ($\Delta\mathbf{AC}$ is the attribute clustering over the value set $\Delta\mathbf{V}$ of the corresponding inserted or deleted tuple t), and iii) $s \in \{\text{inserted, deleted}\}$ indicates whether the corresponding tuple has been inserted or deleted. Notice that for each $A \in \Delta\mathbf{C}$, we have $v \in \mathbf{V}_{i,A} \subseteq \mathbf{V}_i$.

Capturing all data changes in $\mathbf{D}$ (insertions, deletions, modifications of tuples) and extracting them in the form required by the messaging queues can be implemented by database triggers or by the parsing of the transaction log [Kleppmann, 2016]. For instance, LinkedIn's Databus [Das et al., 2012]

and Facebook's Wormhole [Sharma et al., 2015] use the technique based on reading the transaction log at a large scale.

The proposed architecture is highly configurable. For each three components (Q_i, N_i, DB_i) $(1 \leq i \leq n)$, we have more than one option regarding the number of machines on which they can be installed: i) Each of them can have its own machine, ii) each two components can be installed on one machine, or iii) the three components can share a machine. Moreover, it might not be necessarily for a messaging queue to be assigned to only one node–this means that a massaging queue can deliver more than one computation node. On the other hand, a computation node can also receive events from more than one massaging queue. We also have more than one possibility for the relationship between a computation node and the database of the corresponding generator: More than one generator can be managed by a database and more than one attribute clustering can be managed by a computation node. The appropriate configuration depends on the choice the frameworks and on the workload of the database.

To decide if $A \subseteq B$ is valid for two attributes $A, B \in \mathbf{A}$, we apply Lemma 2.8 in Chapter 3 or Lemmas 2.6 and 2.3 together.

Comparing with data-intensive applications When a user posts content to Facebook, the content is written into a database. There are many Facebook's applications interested in such newly-added content in time, since they need to make updates immediately. For instance, News Feed is interested in the update to be able to serve new stories to the friends of the user who just posted. Similarly, users receiving a notification might wish to immediately view the new content. A number of internal services, such as index server pipeline, cache invalidation pipeline, are also interested in every update to the databases [Sharma et al., 2015]. Wormhole is a publish-subscribe system built and deployed by Facebook to identify new posts and to deliver updates to all associated applications. Wormhole directly reads the transaction log maintained by the database to identify the new committed posts. Wormhole is also based on dividing the user generated data into a number of disjoint subsets for better scaling. The idea of mining the database log to capture the changes in the data is also implemented by LinkedIn [Das et al., 2012] to update a wide range of downstream applications as soon as possible.

3.8 Experimental evaluation

We now evaluate our system in terms of Definition 3.1, which means that we experimentally investigate the runtime T_{inc} needed to update the data structures after inserting a tuple into the corresponding dataset and after deleting a tuple from it. As updating the attribute clustering after a modification of an existing tuple can be reduced to a composite operation consisting of an update after an insertion and an update after a deletion (see Example 3.5), there is no need for conducting experiments for estimating the average time needed after a modification of a tuple. In this section, we often refer to the expression "updating the data structures of the system" as "updating the attribute clustering".

In particular, we carry out this evaluation to answer the following questions:

Table 3.2 Characteristics of datasets used in the experiments

D	$\|\mathbf{D}\|$	$\|\mathbf{A}\|$	$\sum_{r_i \in \mathbf{D}} \|r_i\|$	$\|\mathbf{V}\|$	$\|\mathbf{AC}\|$	$\|\mathbf{I}\|$
TPC-H	8	61	8,661,245	11,807,306	126	80
MB1	45	273	10,000,000	10,382,340	663,584	1844
MB2	18	100	24,000,000	20,552,799	294,059	178
PLISTA1	4	140	33,364,151	46,882,120	185	408
H-GENOME	43	387	116,227,014	72,559,365	287,738	4976

1. What is the average runtime T_{inc} for updating the attribute clustering of different large datasets?
2. Can we ignore the incremental runtime toward the runtime required by the static discovery of uINDs?
3. How effective are the cache strategies of the system?
4. How does the incremental runtime T_{inc} change in relation to an increase in the number of attributes?
5. How does the incremental runtime T_{inc} change in correlation with the growth in the number of tuples?

The function of the quantities F_A^v ($A \in \mathbf{A}$ and $v \in \mathbf{V}$) in the data structure of the generator is to handle the deletion. Thus, we do not need these quantities if we limit our system to support only the insert. Therefore, the following question arises: how does the updating time for the insertion change if we limit our system to support only the insertion? To answer this question, we implemented an extra version of Algorithm 8, in which the generator does not contain the frequency of occurrences F_A^v. We refer to this version of the implementation as `only-insert`.

3.8.1 Setup

Experimental conditions We performed the experiments on a Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We installed Oracle 11g on the same machine as the database server and used an external disk for the storage of all used datasets and for the storage of the generators. We implemented all algorithms required for the experiments in 64-bit Java 7.

Datasets Table 3.2 shows some characteristics of the datasets used in our experiments. The first column states the name of the dataset. The second column gives the number of relations in the corresponding dataset. The total number of attributes in each dataset is given in the third column. The

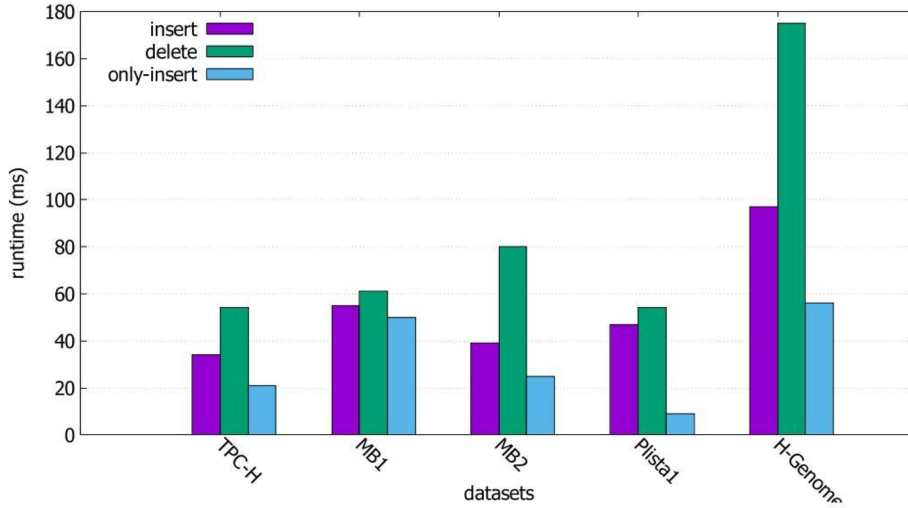


Figure 3.4 Comparing the runtime for an insert with the runtime for a delete

fourth column states the total number of rows in each dataset. The number of distinct values in each dataset is given in the fifth column. The number of clusters in the attribute clustering of each dataset is stated in the sixth column. The last column gives the number of valid uINDs in each dataset.

TPC-H is a benchmark dataset available at http://tpc.org/tpch. All the other datasets are real-world datasets. Both MB1 and MB2 are subsets of MUSICBRAINZ dataset available at https://musicbrainz.org. MUSICBRAINZ is an open music encyclopedia that collects music metadata and makes them available to the public. PLISTA1 is a subset of the dataset PLISTA [Kille et al., 2013], which contains anonymized web-log data provided by the advertisement company Plista. H-GENOME is a genome dataset of homo sapiens available at http://ensembl.org.

3.8.2 Evaluation of the performance

To estimate the time T_{inc} needed for updating the attribute clustering of different datasets, we designed the experiments as follows:

Design of experiments From each dataset $\mathbf{D}$ in Table 3.2, we randomly selected two different sets of tuples $\mathbf{D}_{ins}$ and $\mathbf{D}_{del}$. Random selection of tuples for the experiments helps to avoid selecting tuples having a lot of values in common. If the selected tuples share a lot of values, then the capability of the caches increases, which reduces the average time needed for updating the clusters. Each selected set consists of around 100,000 tuples. The set $\mathbf{D}_{ins}$ is for conducting experiments for insertions, while $\mathbf{D}_{del}$ is for conducting experiments for deletions. Then, we removed all tuples of $\mathbf{D}_{ins}$ from the original dataset $\mathbf{D}$ to reinsert them again in a later step. After that, we initialized the data structures *ACRecs* and *GenRecs* for $\mathbf{D} \setminus \mathbf{D}_{ins}$ by applying Algorithm 14. For each dataset, the statistics presented in Table 3.2 are calculated for $\mathbf{D} \setminus \mathbf{D}_{ins}$.

Table 3.3 The reduction of generator accesses gained by cache strategies

D	reduction by insert(%)	reduction by delete(%)
TPC-H	99.987	99.984
MB1	73.440	81.213
MB2	93.177	83.896
PLISTA1	99.997	99.995
H-GENOME	90.076	91.811

To estimate the time for updating the attribute clustering of **D** after inserting a tuple, we insert all the tuples of $\mathbf{D}_{ins}$ again into **D**. After each insertion, we updated the data structures. For each update, we recorded the needed time and took the average of all runtimes. We repeated this procedure for the tuples $\mathbf{D}_{del}$, but instead of insertions, we removed all tuples of $\mathbf{D}_{del}$ from **D**.

For the only-insert version we recreated and reinitialized the *GenRecs* without the quantities F_A^v for each dataset **D** and used the same set $\mathbf{D}_{ins}$ selected for the regular version.

Figure 3.4 shows the results of these experiments. In this figure, there is a group of three bars for each dataset. In each group, the left bar presents the average time required to update the attribute clustering after inserting a tuple, while the middle bar presents the average time needed to update the attribute clustering after a deletion. The right bar in each group shows the average time needed to update the attribute clustering after an insertion in the only-insert version.

Evaluation of the runtime For each dataset, the average runtime for updating the attribute clustering after an insertion is shorter than the average runtime for updating the attribute clustering after a deletion. The main reason is that the update after a deletion always needs one more access to the external data structure *GenRecs* than the update after an insertion, as discussed in Subsection 3.5.4. Furthermore, it is clear that the runtime for updating after an insertion in the only-insert version is always less than the runtime for updating after an insertion in the regular version because *GenRecs* in the only-insert version does not contain the quantities F_A^v.

The cost of updating the data structure of the generator depends on two variables: the umber of distinct values $|\mathbf{V}|$ and the number of attributes $|\mathbf{A}|$, which means that the runtime for updating the generator increases if the number of distinct values or the number of attribute increase. This fact explains why the three bars of the dataset H-GENOME are longer than the corresponding bars of the other datasets (see also Table 3.2).

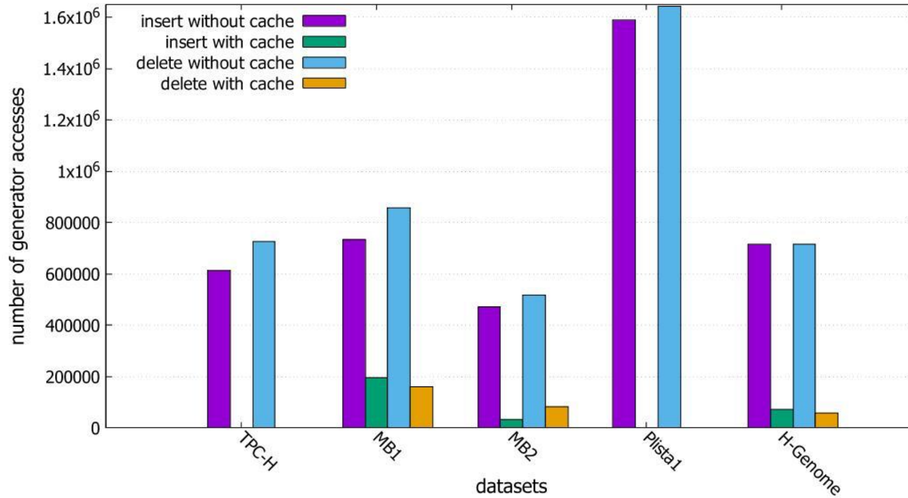


Figure 3.5 Comparing the number of generator accesses in two cases: using the cache strategies and not using the cache strategies.

3.8.3 Evaluation of cache strategies

To evaluate the effectiveness of our cache strategies, we compared the number of generator accesses in the case of using both data structures *CountCache* and *ValCache* with the required number of generator accesses without using them. In both cases, we counted the number of accesses needed to insert the set $\mathbf{D}_{ins}$ and the number of accesses needed to delete the $\mathbf{D}_{del}$ of each dataset in Table 3.2. The results are presented in Figure 3.5. In this figure, there is a group of four bars for each dataset. Form left to right in each group, the first bar presents the number of accesses for inserting the $\mathbf{D}_{ins}$ without caches, while the second bar is for the same insertions but with caches. The third bar presents the number of accesses for deleting $\mathbf{D}_{del}$ without caches, while the far right bar is for the same deletions but with caches. Table 3.3 shows the percentage of reducing the number of accessing the generator if the caches are used.

Overall, we reduced the number of generator accesses by more than 73 % for the insertion, and by more than 81 % for the deletion. For TPC-H and PLISTA1, the reduction is more than 99 % both in the insertion and in the deletion. This high reduction explains why the bars presenting the number of accesses in the case of using the caches do not appear in Figure 3.5 for both TPC-H and PLISTA1.

3.8.4 Comparing with the static discovery

We now compare the incremental discovery with the static discovery of uINDs in terms of Definition 3.1. For that purpose, we compared each runtime calculated in the previous subsection for each dataset in Table 3.2 with the corresponding runtime required by S-INDD [Shaabani and Meinel, 2015]. These comparisons are presented in Table 3.4. As we can observe, the runtime needed by the incremental update of the attribute clustering is much smaller than the runtime needed by S-INDD. This observation is valid for all datasets. For instance, after inserting a tuple t into H-GENOME,

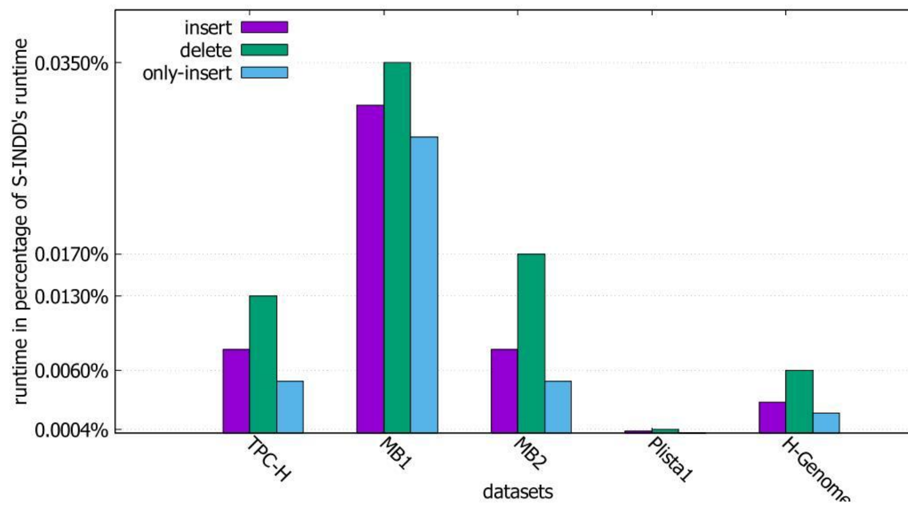


Figure 3.6 Runtime in percentage of S-INDD runtime

Table 3.4 Comparing the runtime (in seconds) with the runtime of S-INDD [Shaabani and Meinel, 2015] applied to the entire dataset

D	S-INDD	only-insert	insert	delete
TPC-H	424	0.021	0.034	0.054
MB1	176	0.050	0.055	0.061
MB2	484	0.025	0.039	0.080
PLISTA1	13,580	0.009	0.047	0.054
H-GENOME	3135	0.056	0.097	0.175

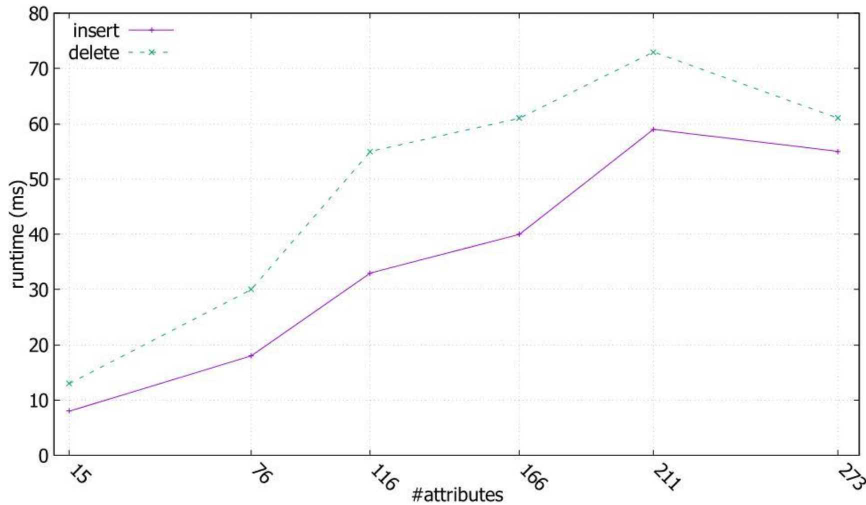


Figure 3.7 Scaling the number of attributes and fixing the number of rows to 10,000,000

S-INDD needs $T_{\text{S-INDD}}(\text{H-GENOME} + \{t\}) = 3135$ seconds, while the incremental approach needed only $T_{inc}(t) = 0.097$ seconds (i.e., the incremental approach needed ca. 0.003 % of the time needed by S-INDD). Thus, we can neglect $T_{inc}(t) = 0.097$ seconds toward $T_{\text{S-INDD}}(\text{H-GENOME} + \{t\}) = 3135$ seconds. For updating the attribute clustering of PLISTA1 after a deletion, the incremental approach requires only 0.0004 % of the runtime required by S-INDD (i.e., the reduction in the static runtime is more than 99.9996 %) here. In fact, we can neglect each incremental runtime listed in Table 3.4 toward the corresponding statical runtime, which means that our system satisfies the requirement formulated in Definition 3.1 for incremental discovery of uINDs.

Figure 3.6 presents the incremental runtime in percentage of the corresponding static runtime. This presentation allows us to make an important observation, namely, that there is a tendency for the percentages of bigger datasets (PLISTA1 and H-GENOME) to be smaller than those of the smaller datasets. That means that by incrementally updating the uINDs, we avoid the performance suffering from static rediscovery after a simple change in the corresponding dataset.

We have to mention that in these experiments S-INDD used the sorting function of the underlying DBMS for sorting the value sets of the attributes, which explains why S-INDD is here faster in processing the dataset H-GENOME than processing it in the experiments of Chapter 2.

If we used the incremental updates for the static discovery, then there would be a significant overhead. For instance, if we applied the version of the incremental discovery to the entire dataset PLISTA1 or to the entire dataset TPC-H, then the S-INDD would be faster by the factor of 20 for PLISTA1 and by the factor of 430 for TPC-H.

The purpose of incrementally updating uINDs, however, is not to replace the static discovery of uINDs, but to update them in the case of any change in the data. Therefore, we developed the approach in Subsection 3.5.5 for initializing the data structures of the incremental discovery to avoid initializing them incrementally.

3.8.5 Scaling the number of attributes

We now evaluate the change in the runtime in relation to the increase in the number of attributes. For this purpose, we designed the experiments in this case as follows.

Design of experiments For these experiments, we created six datasets $\mathbf{D}^i$ $(1 \leq i \leq 6)$ from the dataset MUSICBRAINZ, with the following properties: (i) $\mathbf{D}^6 = \mathrm{MB1}$, (ii) $\mathbf{D}^1$ has 15 attributes, (iii) the dataset $\mathbf{D}^{i+1}$ contains all relations of $\mathbf{D}^i$ and additional relations from MUSICBRAINZ so that $\mathbf{D}^{i+1}$ has around 50 attributes more than $\mathbf{D}^i$ $(1 \leq i \leq 5)$, and (iv) each $\mathbf{D}^i$ $(1 \leq i \leq 6)$ has around 10,000,000 tuples. For each dataset $\mathbf{D}^i$, we estimated the runtime for updating the attribute clustering by applying the same process applied in Subsection 3.8.2. Figure 3.7 shows the results of these experiments.

Evaluation In this figure, we can observe a tendency for the incremental runtime to increase when the number of attributes increases, because (i) the size of the generator grows when the number of the attributes grows, and (ii) the runtime for the modification of the generator correlates with its size. The reason for the correlation is that the time of creating and executing the SQL-Statements needed to update the generator generally increases when the number of the attribute increases. However, the incremental runtime declines slightly when the number of attributes increases from 211 to 273. This deviation from the general tendency may be because (i) the number of generator accesses for the dataset $\mathbf{D}^5$ (i.e. the dataset with 211 attributes) is higher than the number of generator accesses for $\mathbf{D}^6$ (i.e. the dataset with 273 attributes), and (ii) up to certain degree, the number of generator accesses has a stronger influence on the runtime than the number of attributes does. In fact, the number of accesses for $\mathbf{D}^5$ is 199,808 for the insertions and 174,786 accesses for the deletions, while for $\mathbf{D}^6$, it is 195,340 for the insertions and 161,144 accesses for the deletions.

As we can see, the maximum incremental runtime for the insertion is 0.059 seconds, while the maximum incremental runtime for the deletion is 0.074 seconds, and the static runtime for discovering uINDs in $\mathbf{D}^6 = \mathrm{MB1}$ is 176 seconds (see Table 3.4). Thus, we can ignore both incremental runtimes towards the static runtime.

3.8.6 Scaling the number of tuples

We now evaluate the change in the runtime in correlation with the growth in the number of tuples. Notice that an increase in the number of tuples normally causes an increase in the number of distinct values of the dataset. To achieve this evaluation, we designed the experiments here as follows:

Design of experiments We divided the dataset MB2 (see Table 3.2) into 11 subsets $\mathbf{D}^i$ $(1 \leq i \leq 11)$ with the following properties: (i) $\mathbf{D}^{11} = \mathrm{MB2}$, (ii) $\mathbf{D}^i \subset \mathbf{D}^{i+1}$ $(1 \leq i \leq 10)$, (iii) the dataset $\mathbf{D}^{i+1}$ contains around 2,000,000 tuples more than the dataset $\mathbf{D}^i$ $(1 \leq i \leq 10)$, and (iv) the number of tuples in $\mathbf{D}^1$ is 4,000,000 tuples. Then, for each dataset $\mathbf{D}^i$, we estimated the runtime for updating the

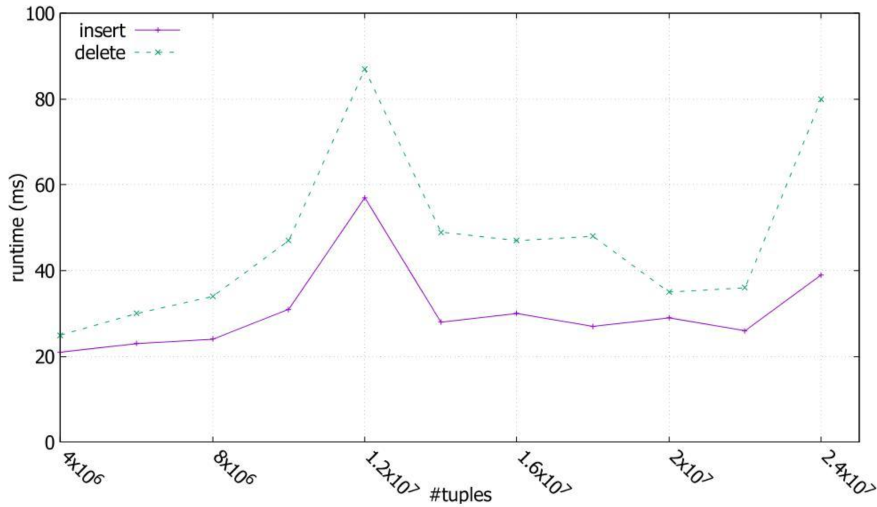


Figure 3.8 Scaling the number of rows and fixing the number of attributes to 100

attribute clustering by following the same process described in Subsection 3.8.2. Figure 3.8 shows the results of these experiments.

Evaluation We observe that there is no unique tendency for the runtime to rise when the number of tuples rises. When the number of tuples increases from 4,000,000 to 10,000,000, the runtime increases slightly. That is in contrast to increasing the number of tuples from 10,000,000 to 12,000,000, where the runtime noticeably increases. It decreases when the number of tuples decreases from 12,000,000 to 14,000,000. Thus, the runtime reaches a peak for the dataset $\mathbf{D}^5$ with 12,000,000 tuples. The reason for this peak is the fact that for $x \in \{ins, del\}$, the number of distinct values in the set $\mathbf{D}^5_x \setminus \mathbf{D}^5$ is much higher than the number of distinct values in $\mathbf{D}^i_x \setminus \mathbf{D}^i$ where $i \neq 5$ and $1 \leq i \leq 10$ respectively. That causes the system to insert, in the case of handling the insertions, more value records into the generator for $\mathbf{D}^5$ than for the other datasets. In the case of handling the deletions, it causes the system to delete more value recodes from the generator for $\mathbf{D}^5$ than for the other datasets. As we discussed in Subsection 3.5.4, deleting records from the generator and inserting records into it are costlier than the modification of its records respectively. Coming back to the trend of the runtime in Figure 3.8, the runtime for the deletions continues to decline slightly until the number of tuples becomes 22,000,000. The runtime for the insertions remains almost constant when the number of tuples increases from 14,000,000 to 24,000,000. Thus, we conclude that the runtime of incrementally detecting uINDs does not necessary increase when the number of tuples increases.

As we can see in Figure 3.8, the maximum runtime for the insertion is 0.057 seconds, while the runtime for the deletion is 0.087 seconds. Thus, we can ignore both incremental runtimes toward the static runtime, which is, according to Table 3.4, 484 seconds for $\mathbf{D}^{10} = \text{MB2}$ (S-INDD has almost the same performance for both datasets $\mathbf{D}^5$ and $\mathbf{D}^{10}$).

3.9 Related work

The need for incremental data profiling methods has been explicitly addressed by Naumann [2013] and Abedjan et al. [2015] to maintain the metadata under data updates without frequently re-profiling the entire dataset. The works of Abedjan et al. [2014] and Wang et al. [2003] are the first ventures into this research area. The former work suggested an approach for unique column combinations discovery in dynamic datasets, while the later work presented an approach for maintaining discovered functional dependencies after data deletions.

From the point of view of data cleaning, Fan et al. [2012] developed methods for incremental detection of any violation of functional dependencies and conditional functional dependencies, while Cong et al. [2007] presented solutions for incremental data repairing with respect to functional dependencies and conditional functional dependencies. Another example of incrementally updating derived data after updating the basic relations is the maintenance of materialized views [Gupta and Mumick, 1995; Gupta et al., 1993]. An example for data mining is the techniques presented by Tsai et al. [1999] for incrementally updating discovered association rules.

It is worth mentioning that most commercial relational DBMSs allow users to specify a set of inclusion dependencies in terms of foreign key constraints between relations. The DBMS validates all user-defined foreign key constraints after an insertion, a deletion, or a change of a tuple and aborts the operation if the result of the update does not satisfy one of these constraints. The DBMS, however, can not find new inclusion dependencies after inserting new tuples or deleting existing ones.

In summary, our work in this chapter is the first one to address the incremental discovery of inclusion dependencies in dynamic datasets.

3.10 Conclusion and future work

In this work, we developed the first approach for incrementally discovering unary inclusion dependencies in frequently changing data. We reduced the problem of the incremental update of uINDs to the incremental update of the attribute clustering, from which unary inclusion dependencies are efficiently derivable. We solved the problem of incrementally updating the attribute clustering by developing new cluster operations to be applied after every data change. Next, we designed algorithms and data structures for an efficient implementation of the cluster operations. The main goal of the incremental discovery is to avoid reprocessing the entire dataset by applying the static discovery after every change–this requires a long time for computation. In this regard, we performed a comprehensive experimental evaluation, which showed that the computation time of the incremental discovery is negligible compared to that of the static discovery. In fact, the reduction in time is up to 99.9996 %.

We also showed how to initialize the data structures to start the incremental discovery with a legacy dataset. Moreover, we suggested a novel algorithm for both the incremental detection and the static detection of approximate uINDs by utilizing the properties of designed data structures.

A shared-nothing architecture for scaling out the incremental discovery of unary INDs was also suggested.

The data structure of the generator is sparse because it maintains all variables F_A^v $(A \in \mathbf{A} \wedge v \in \mathbf{V})$ regardless whether their values are zero or not. Moreover, the implementation of the generator as a relational table is technically limited by the underlying DBMS because a DBMS does not allow the creation of a table whose number of columns exceeds a maximum number defined by the DBMS. A workaround solution to this issue is to create $\lceil |\mathbf{A}|/maxcol \rceil$ tables where *maxcol* is the maximum number of table columns allowed by the corresponding DBMS. This solution significantly increases the space complexity. Moreover, it may increase the number of generator accesses and therefore may decrease the performance of the system. So, a more efficient data structure contains only the variables F_A^v with $F_A^v > 0$ and does not impose restrictions on the number of attributes. An example of such a data structure might be a distributed hash table.

In contrast to the data structure of the generator, that of the clusters is not persistent–it resides constantly in the main memory. Therefore, a system failure causes the loss of the cluster records (i.e. *ACRecs*). Nevertheless, to recover the clusters, we do not need to reinitialize the data structures of the system by applying Algorithm 14 presented in Subsection 3.5.5 because we can efficiently regenerate the cluster records from the data structure of the generator by applying SQL queries. That idea might be the core idea of the implementation of an appropriate recovering strategy.

An inclusion dependency is a necessary but not a sufficient characteristic for a foreign key. This is because the inclusion between value sets may occur by pure chance. To distinguish INDs that are likely to be foreign keys from those that are unlikely to be so, different heuristic approaches have been suggested in [Koeller and Rundensteiner, 2004; Rostin et al., 2009; Zhang et al., 2010]. It is useful to study how we can utilize the data structures of our system to implement these approaches to enable a dynamic ranking of the discovered uINDs according to their likelihood of being foreign keys. Then, the user can periodically explore the incrementally discovered and ranked INDs to progressively understand their dynamic data.

Chapter 4

Discovering Maximum Inclusion Dependencies without Candidate Generation

4.1 Problem statement

As we have discussed in Subsection 1.8.1 of Chapter 1, the existing algorithms for exhaustively discovering all n-ary INDs (n > 1)–i.e., MID [DeMarchi et al., 2002, 2009], ZIGZAG [DeMarchi and Petit, 2003], and FIND_2 [Koeller and Rundensteiner, 2002, 2003] apply the projection invariance of INDs [Casanova et al., 1984; Liu et al., 2012]: A valid n-ary IND implies sets of k-ary valid INDs ($1 \leq k \leq n$). Thus, the number of all valid INDs implied by a valid n-ary IND is 2^n.

MIND implements the generate-and-test (or the levelwise) approach, which is illustrated by Algorithm 1 in Subsection 1.8.1 of Chapter 1, by straightforward adaptation of the Apriori-Algorithm for mining frequent itemsets [Agrawal and Srikant, 1994]. For discovering a single valid IND σ of size n, the Apriori-based approach has to discover $2^n - 1$ implied INDs before even considering σ. Thus, MIND has to execute 2^n *SQL* queries for the validation. Experiments conducted by DeMarchi and Petit [2003]; Koeller and Rundensteiner [2002]; Papenbrock et al. [2015] have shown that Apriori-based algorithms do not scale beyond the maximum IND size of 8.

Attempting to reduce the exponential number of database accesses needed by the Apriori-based approach, FIND_2 and ZIGZAG transform the IND discovery problem into a discovery problem in a hypergraph whose nodes are all valid unary INDs. FIND_2 maps the IND-discovery problem to the hyperclique-discovery problem, while ZIGZAG maps it to the minimal-traversal-discovery problem. Both problems are polynomial in the number of edges and therefore exponential in terms of the number of nodes in the hypergraph because the number of edges in a hypergraph of n nodes is bounded by 2^n. In principle, both algorithms first discover unary and binary INDs by enumeration and validation. Then, they optimistically assume that all high-arity INDs constructed from validated unary

and binary INDs (or in general, from validated INDs in the previous iteration) are likely to be valid. The assumption makes both algorithms extremely sensitive to an overestimation of valid unary and binary INDs–a high number of such small INDs can cause many invalid larger IND candidates being generated and validated against the database. Furthermore, hypergraph-based algorithms have high complexity and are scalable only for sparse hypergraphs [Koeller and Rundensteiner, 2002, 2004].

Consequently, the research question addressed in this chapter is how we can discover all valid n-ary INDs ($n > 1$) without generating candidates and testing them against the dataset.

Contributions. We answered this research question by devising a novel discovery algorithm called MIND$_2$ [Shaabani and Meinel, 2016]. MIND$_2$ (short for **M**aximum **IN**clusion **D**ependency **D**iscovery) discovers all maximum INDs without any candidate generation, where a maximum IND is a valid n-ary IND that can not be implied by any other valid IND.

MIND$_2$ introduces novel characterizations of the maximum INDs. These characterizations are based on operations defined on new metadata called unary IND coordinates, which MIND$_2$ generates by accessing the datasets only $2 \times$ the number of valid unary INDs. So, MIND$_2$ eliminates the exponential number of data accesses needed by the other approaches. In particular, we made the following contributions:

- We introduced the concept of the unary IND coordinates and showed how to infer the set of maximum INDs from them. This inference presents the novel characterizations of the maximum inclusion dependencies.
- We proposed data structures and algorithms to organize and generate the unary IND coordinates and to implement the inference of the maximum INDs from the coordinates. The number of database accesses needed for the generation of the coordinates is linear in the number of valid unary INDs ($2 \times$ the number of valid unary INDs).
- We compared the performance of MIND$_2$ with that of FIND$_2$ using real and synthetic datasets. The experiments showed that MIND$_2$ is faster than FIND$_2$. Furthermore, they showed that MIND$_2$'s scalability, in contrast to that of FIND$_2$, is not influenced by a high number of small valid unary INDs.

The rest of this chapter is organized as follows: Section 4.2 defines the maximum INDs based on the set presentation of the INDs. Section 4.3 formulates our novel rules for inferring the maximum INDs without candidate generation and proves their correctness. Section 4.4 presents MIND$_2$, a possible implementation of the mathematical rules developed in the previous section. The results of an experimental evaluation of MIND$_2$'s performance, compared with that of FIND$_2$, are reported in Section 4.5. The related work is discussed in Section 4.6 in more details. Section 4.7 concludes and discusses future works in connection with another implementation of the principles developed in Section 4.3.

4.2 Maximum inclusion dependency

Let $R[A_1,\ldots,A_{|R|}], S[B_1,\ldots,B_{|S|}] \in \mathbf{R}$ be two relational schemas with the corresponding relations $r,s \in \mathbf{D}$ over $\mathbf{R}$. We present every n-ary IND $\sigma = R[X] \subseteq S[Y] \in \Sigma_{R\to S}$[1] with $X = [A_{i_1},\ldots,A_{i_n}]$ and $Y = [B_{i_1},\ldots,B_{i_n}]$ as a set of all unary INDs $A_{i_k} \subseteq B_{i_k}$ $(1 \leq i_k \leq i_n)$. That is,

$$\sigma = \{A_{i_1} \subseteq B_{i_1},\ldots,A_{i_n} \subseteq B_{i_n}\} \tag{4.1}$$

Furthermore, we identify the set of all attributes occurring on the left hand side of σ with $LHS(\sigma)$ and the set of all attributes occurring on the right hand side of σ with $RHS(\sigma)$. Thus, we have

$$LHS(\sigma) = \{A_{i_1},\ldots,A_{i_n}\} \tag{4.2}$$

$$RHS(\sigma) = \{B_{i_1},\ldots,B_{i_n}\} \tag{4.3}$$

Now, we introduce the concept of a maximum IND based on the set presentation of INDs.

Definition 4.1. *(Maximum IND) Let $I \in \Sigma_{r\to s}$ be a valid IND over R and S. I is a maximum IND if and only if there is no $I' \in \Sigma_{r\to s}$ such that $I \subset I'$ holds. We denote the set of all maximum INDs between R and S according to r and s with $\mathbf{I}_M(\Sigma_{r\to s})$, or with $\mathbf{I}_M$ if $\Sigma_{r\to s}$ is understood. That is,*

$$\mathbf{I}_M(\Sigma_{r\to s}) = \{I \in \Sigma_{r\to s} \mid \neg\exists I' \in \Sigma_{r\to s} : I \subseteq I'\} \tag{4.4}$$

Notice that the set $\mathbf{I}_M(\Sigma_{r\to s})$ is equal to the set $Bd^+(\Sigma_{r\to s})$ defined by Equation 1.15 in Section 1.8.

Having $\mathbf{I}_M(\Sigma_{r\to s})$ computed, we can derive the set $\Sigma_{r\to s}$ based on the projection inference-rule presented in Section 1.7 as follows:

$$\Sigma_{r\to s} = \{\sigma \mid \exists M \in \mathbf{I}_M(\Sigma_{r\to s}) : \sigma \subseteq M\} \tag{4.5}$$

Thus, The set $\mathbf{I}_M(\Sigma_{r\to s})$ can be considered as a concise representation of the set $\Sigma_{r\to s}$. Therefore, our goal in this chapter is to directly compute $\mathbf{I}_M(\Sigma_{r\to s})$ without any intermediate IND sets (candidates). In the rest of the chapter, we refer to the set $\mathbf{I}_M(\Sigma_{r\to s})$ simply as $\mathbf{I}_M$.

Example 4.1. *According to the two relations presented in Table 4.1, the set of all valid unary INDs over R and S is*

$$\mathbf{I}_{r\to s} = \{u_i = A_i \subseteq B_i \mid 1 \leq i \leq 5\}$$

The set of all n-ary valid INDs over R and S is

$$\Sigma_{r\to s} = \{\{u_1,u_2\},\{u_1,u_3\},\{u_2,u_3\},\{u_1,u_2,u_3\},\{u_1,u_4\},\{u_2,u_4\},\{u_1,u_2,u_4\},\{u_4,u_5\}\}$$

[1] See Section 1.5 of Chapter 1 to recall some basic notions

Table 4.1 Running example

R

ID_R	A_1	A_2	A_3	A_4	A_5
1	a	b	c	d	e
2	f	g	i	j	k

S

ID_S	B_1	B_2	B_3	B_4	B_5
1	a	b	c	d	$\perp$
2	$\perp$	$\perp$	c	d	$\perp$
3	$\perp$	$\perp$	c	d	e
4	f	g	i	$\perp$	$\perp$
5	f	g	$\perp$	j	k

The set of all maximum INDs over R and S is

$$\mathbf{I}_M = \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_4, u_5\}\}$$

For example, from the maximum IND $M = \{u_1, u_2, u_3\} \in \mathbf{I}_M$ we can conclude that the n-ary INDs $\{u_1, u_2\}$, $\{u_2, u_3\}$, $\{u_1, u_2\}$, and M itself are included in $\Sigma_{r \to s}$.

4.3 Principles of MIND$_2$

The mathematical framework for inferring the maximum INDs without candidate generation is based on three principles formulated in this section.

4.3.1 Principle 1

For every tuple pair $r_i \in r$ and $s_j \in s$, we compute M^{ij}, the maximum IND between $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$ according to r_i and s_j ($1 \leq i \leq |r|$ and $1 \leq j \leq |s|$). To characterize the set M^{ij}, we introduce two new concepts: Attribute value-positions and valid unary IND coordinates.

Definition 4.2. *(Attribute value-positions) The value positions of an attribute $A \in U$ with $U \in \{R, S\}$ is the set*

$$P_A = \pi_{\{ID_U, A\}}(U) \tag{4.6}$$

Definition 4.3. *(Unary IND coordinates) The coordinates of a valid unary IND $u \in \mathbf{I}_{r \to s}$ is the set*

$$C_u = \{(i, j) \mid \exists(i, v) \in P_{LHS(u)} \wedge \exists(j, v') \in P_{RHS(u)} : v = v'\} \tag{4.7}$$

The coordinates of a valid unary IND $u \in \mathbf{I}_{r \to s}$ is the set of all tuple-ID pairs (i, j) where the value of the attribute $LHS(u)$ in the tuple $r_i \in r$ is identical with the value of the attribute $RHS(u)$ in the tuple $s_j \in s$. Hence,

$$(i, j) \in C_u \text{ if and only if } r_i[LHS(u)] = s_j[RHS(u)]$$

Table 4.2 The coordinates of all valid uINDs between R and S in presented Table 4.1

i	P_{A_i}	P_{B_i}	$C_{A_i \subseteq B_i}$
1	$\{(1,a),(2,f)\}$	$\{(1,a),(2,\bot),(3,\bot),(4,f),(5,f)\}$	$\{(1,1),(2,4),(2,5)\}$
2	$\{(1,b),(2,g)\}$	$\{(1,b),(2,\bot),(3,\bot),(4,g),(5,g)\}$	$\{(1,1),(2,4),(2,5)\}$
3	$\{(1,c),(2,i)\}$	$\{(1,c),(2,c),(3,c),(4,i),(5,\bot)\}$	$\{(1,1),(1,2),(1,3),(2,4)\}$
4	$\{(1,d),(2,j)\}$	$\{(1,d),(2,d),(3,d),(4,\bot),(5,j)\}$	$\{(1,1),(1,2),(1,3),(2,5)\}$
5	$\{(1,e),(2,k)\}$	$\{(1,\bot),(2,\bot),(3,e),(4,\bot),(5,k)\}$	$\{(1,3),(2,5)\}$

Having the coordinates of all unary INDs generated, we can compute the maximum IND M^{ij} between any tuple pair (r_i, s_j) without a database access based on the following lemma.

Lemma 4.1. *M^{ij} consists of all unary INDs $u \in \mathbf{I}_{r\to s}$ with $(i,j) \in C_u$. That is,*

$$M^{ij} = \{u \in \mathbf{I}_{r\to s} \mid (i,j) \in C_u\} \tag{4.8}$$

Proof. Let $M^{ij} = \{u_1,\ldots,u_n\}$ be the set of all valid uINDs with $(i,j) \in C_{u_k}$ and $1 \le k \le n$. According to Definition 4.3,

$$(\forall k \in \{1,\ldots,n\})(\exists (i,v_k) \in P_{LHS(u_k)} \wedge \exists (j,v'_k) \in P_{RHS(u_k)}) : v_k = v'_k$$

Thus,

$$(v_1,\ldots,v_k,\ldots,v_n) = (v'_1,\ldots,v_k,\ldots,v'_n)$$

That means that

$$r_i[LHS(M^{ij})] = s_j[RHS(M^{ij})]$$

Hence, M^{ij} is a valid IND over $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$ according to $\{r_i, s_j\}$ (see Definition 1.1).

We now have to show that M^{ij} is maximum. We assume that M^{ij} is not maximum. That means that, based on Definition 4.1,

$$\exists M_1^{ij} \in \mathbf{I}_M : M^{ij} \subset M_1^{ij}$$

Hence,

$$\exists u' \in \mathbf{I}_{r\to s} : u' \in M_1^{ij} \wedge (i,j) \notin C_{u'}$$

That means that the value of the attribute $LHS(u')$ in r_i is different from the value of the attribute $RHS(u')$ in s_j. Therefore, $r_i[LHS(M_1^{ij})] \neq s_j[RHS(M_1^{ij})]$, which means that M_1^{ij} is not valid. Thus, our assumption is wrong. □

Example 4.2. *Based on our running example, the second column in Table 4.2 lists the value positions P_{A_i} of R's attributes while the value positions P_{B_i} of S's attributes are listed in the third column. The last column in this table shows the coordinates of all valid unary INDs between R and S (see Example 4.1). For example, for $A_5 \subseteq B_5$, we have*

$$(1,e) \in P_{A_5} \text{ and } (3,e) \in P_{B_5}$$

Therefore,

$$(1,3) \in C_{A_5 \subseteq B_5}$$

Also,

$$(2,5) \in C_{A_5 \subseteq B_5}$$

because

$$(2,k) \in P_{A_5} \text{ and } (5,k) \in P_{B_5}$$

The maximum INDs M^{ij} between r_i and s_j ($1 \leq i \leq 2$ and $1 \leq j \leq 5$) are

$$M^{1,1} = \{u_1,u_2,u_3,u_4\}, \quad M^{1,2} = \{u_3,u_4\}, \quad M^{1,3} = \{u_3,u_4,u_5\}, \quad M^{1,4} = M^{1,5} = \emptyset$$
$$M^{2,1} = M^{2,2} = M^{2,3} = \emptyset, \quad M^{2,4} = \{u_1,u_2,u_3\}, \quad M^{2,5} = \{u_1,u_2,u_4,u_5\}$$

Let us explain, for example, the content of the maximum IND $M^{1,2}$ between r_1 and s_2. We have

$$(1,2) \in C_{u_3}$$

Therefore,

$$u_3 \in M^{1,2}$$

Also,

$$u_4 \in M^{1,2}$$

because

$$(1,2) \in C_{u_4}$$

But

$$u_1,u_2,u_5 \notin M^{1,2}$$

because

$$(1,2) \notin C_{u_1}, (1,2) \notin C_{u_2}, \text{ and } (1,2) \notin C_{u_5}$$

In the next step, we compute the set of all maximum INDs between every tuple $r_i \in r$ and the relation s based on the following principle, respectively.

4.3.2 Principle 2

For every tuple $r_i \in r$, we compute $\mathbf{I}_M^i$, the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S according to r_i and s. To characterize the set $\mathbf{I}_M^i$, we introduce the following operator.

Definition 4.4. *(ϕ-operator) The operator $\phi : 2^{\Sigma_{R \to S}} \to 2^{\Sigma_{R \to S}}$ is defined as*

$$\phi(\mathbf{S}) = \{\sigma \in \mathbf{S} \mid \nexists \sigma' \in \mathbf{S} : \sigma \subset \sigma'\} \tag{4.9}$$

That is, the operator ϕ takes a set $\mathbf{S}$ of INDs and returns each IND in $\mathbf{S}$ that is not included in any other IND from $\mathbf{S}$. Thus, we conclude that $\phi(\mathbf{S}) \subseteq \mathbf{S}$ for any $\mathbf{S} \in 2^{\Sigma_{R \to S}}$.

Lemma 4.2. *Let $\mathbf{I}^i$ be the set of all non-empty M^{ij} ($1 \leq j \leq |s|$). Then, the set of all maximum INDs over $\sigma_{ID_R=i}(R)$ and S is obtained by*

$$\mathbf{I}_M^i = \phi(\mathbf{I}^i) \tag{4.10}$$

Proof. Every $M^{ij} \in \mathbf{I}^i$ is a valid but not necessary a maximum IND between $\sigma_{ID_R=i}(R)$ and S. But what we want to have is all maximum INDs from $\mathbf{I}^i$. Based on Definition 4.4, ϕ-operator solves this task. Thus, $\mathbf{I}_M^i = \phi(\mathbf{I}^i)$ is the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S. □

Example 4.3. *Based on Example 4.2, we have*

$$\mathbf{I}^1 = \{M^{1,1}, M^{1,2}, M^{1,3}\}, \qquad \mathbf{I}_M^1 = \phi(\mathbf{I}^1) = \{M^{1,1}, M^{1,3}\}$$
$$\mathbf{I}^2 = \{M^{2,4}, M^{2,5}\}, \qquad \mathbf{I}_M^2 = \phi(\mathbf{I}^2) = \{M^{2,4}, M^{2,5}\}$$

We can now compute $\mathbf{I}_M$, the set of all maximum INDs between R and S, from the sets $\mathbf{I}_M^i$ ($1 \leq i \leq |r|$) based on Principle 3.

4.3.3 Principle 3

To explain the main idea behind Principle 3, let us consider the two relations in Table 4.1. What are the maximum INDs between them if we know $\mathbf{I}_M^1$ and $\mathbf{I}_M^2$ computed in Example 4.3? Firstly, the intersection between any two INDs $M_1 \in \mathbf{I}_M^1$ and $M_2 \in \mathbf{I}_M^2$ is a valid IND between R and S. For example, $M^{1,1} \cap M^{2,4} = \{u_1, u_2, u_3\}$ is a valid IND between R and S. Secondly, after computing the intersection between each pair $(M_1, M_2) \in \mathbf{I}_M^1 \times \mathbf{I}_M^2$, taking all maximum sets from the result gives us the set of all maximum INDs. We generalize these two ideas as follows.

Definition 4.5. *(ψ-operator) The operator $\psi : 2^{\Sigma_{R \to S}} \times 2^{\Sigma_{R \to S}} \to 2^{\Sigma_{R \to S}}$ is defined as*

$$\psi(\mathbf{S}_1, \mathbf{S}_2) = \{\sigma \mid \exists (\sigma_1, \sigma_2) \in \mathbf{S}_1 \times \mathbf{S}_2 : \sigma = \sigma_1 \cap \sigma_2 \neq \emptyset\} \tag{4.11}$$

That is, for two sets $\mathbf{S}_1$ and $\mathbf{S}_2$ of INDs the ψ-operator takes every tuple (σ_1, σ_2) from $\mathbf{S}_1 \times \mathbf{S}_2$ and computes the intersection between σ_1 and σ_2.

To characterize the computation of the set $\mathbf{I}_M$, we define the ρ-operator.

Definition 4.6. *(ρ-operator) Let $\mathbf{I_M}$ be the set of all $\mathbf{I}_M^i$ ($1 \leq i \leq |r|$). The ρ-operator is defined as*

$$\rho(\mathbf{I_M}) = \begin{cases} \mathbf{S} & \text{if } |\mathbf{I_M}| = 1 \text{ and } \mathbf{S} \in \mathbf{I_M} \\ \phi(\psi(\mathbf{S}, \rho(\mathbf{I_M} \setminus \{\mathbf{S}\}))) & \text{if } |\mathbf{I_M}| > 1 \text{ and } \mathbf{S} \in \mathbf{I_M} \end{cases} \tag{4.12}$$

Now, we can compute $\mathbf{I}_M$ as follows.

Lemma 4.3. *The set of all maximum IND over R and S is obtained by*

$$\mathbf{I}_M = \rho(\mathbf{I_M}) \tag{4.13}$$

Proof. We prove the lemma by induction on the number of tuples i in r.

Basis Step: For $i = 1$, we have

$$\mathbf{I_M} = \{\mathbf{I}_M^1\}$$

Thus,

$$\rho(\{\mathbf{I}_M^1\}) = \mathbf{I}_M^1 = \mathbf{I}_M$$

based on the construction of the set $\mathbf{I}_M^1$

Induction Assumption: For $1 \leq i < |r|$, let $\mathbf{I'_M}$ be the set of all $\mathbf{I}_M^i$ and $\mathbf{I}'_M$ be the set of all maximum INDs between $\sigma_{ID_R<|r|}(R)$ and S. We assume

$$\mathbf{I}'_M = \rho(\mathbf{I'_M}) \tag{4.14}$$

Inductive Step: Let $\mathbf{I}_M^{|r|}$ be the set of all maximum INDs between $\sigma_{ID_R=|r|}(R)$ and S. Thus,

$$\mathbf{I_M} = \mathbf{I'_M} \cup \mathbf{I}_M^{|r|} \tag{4.15}$$

We have to show

$$\begin{aligned} \mathbf{I}_M &= \rho(\mathbf{I_M}) \\ &= \rho(\mathbf{I'_M} \cup \mathbf{I}_M^{|r|}) && \text{(based on 4.15)} \\ &= \phi(\psi(\mathbf{I}_M^{|r|}, \rho(\mathbf{I'_M}))) && \text{(based on 4.12)} \\ &= \phi(\psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M)) && \text{(based on 4.14)} \end{aligned}$$

For any valid IND $I \in \mathbf{I}$ over R and S, the following condition holds:

$$(\exists I_1 \in \mathbf{I}_M^{|r|} : I \subseteq I_1) \text{ and } (\exists I_2 \in \mathbf{I}'_M : I \subseteq I_2)$$

Thus,

$$I \subseteq I_1 \cap I_2$$

According to the definition of ψ-operator,

$$J = I_1 \cap I_2 \in \psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M)$$

Thus,

$$(\forall I \in \mathbf{I})(\exists J \in \psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M)) : I \subseteq J$$

Therefore,

$$\mathbf{I}_M \subseteq \psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M)$$

That means that applying the ϕ-operator to the set $\psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M)$ gives us the set $\mathbf{I}_M$. That is,

$$\mathbf{I}_M = \phi(\psi(\mathbf{I}_M^{|r|}, \mathbf{I}'_M))$$

□

Example 4.4. *Based on Example 4.3, we have*

$$\mathbf{I_M} = \{\mathbf{I}_M^1, \mathbf{I}_M^2\}$$

Accordingly,

$$\begin{aligned}\psi(\mathbf{I}_M^1, \mathbf{I}_M^2) &= \{M^{1,1} \cap M^{2,4}, M^{1,1} \cap M^{2,5}, M^{1,3} \cap M^{2,4}, M^{1,3} \cap M^{2,5}\} \\ &= \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_3\}, \{u_4, u_5\}\}\end{aligned}$$

$$\begin{aligned}\mathbf{I}_M &= \rho(\mathbf{I_M}) \\ &= \phi(\psi(\mathbf{I}_M^1, \rho(\{\mathbf{I}_M^2\}))) \\ &= \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_4, u_5\}\} \quad \textit{(cf. Example 4.1)}\end{aligned}$$

In the following section, we formulate MIND$_2$ algorithmically. We also present its data structures. This formulation is the basis of our implementation of MIND$_2$.

4.4 Mind$_2$

4.4.1 Overall workflow

MIND$_2$ consists of three major components. Algorithm 18 as the first component, is responsible for computing the unary IND coordinates C_u of each valid unary IND $u \in \mathbf{I}_{r \to s}$ based on Definition 4.3. It also stores each generated set C_u in a separate file in an external repository *Repo* on a hard drive.

Then, Algorithm 19 reads the generated coordinates at once and computes the set of all maximum INDs $\mathbf{I}_M$ incrementally according to the ascending order of the tuple-IDs $i \in ID_R$ in the left relation r.

In other words, it computes the ρ-operator (Definition 4.6) iteratively. Before the iteration in which the set $\mathbf{I}_M^i$ (the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S) can be generated, Algorithm 19 computes all maximum INDs between $\sigma_{ID_R<i}(R)$ and S and stores them in $\mathbf{I}_M$. In other words, before the computation of $\mathbf{I}_M^i$ starts, the set $\mathbf{I}_M$ contains the maximum INDs between the tuples $\{r_k \in r \mid 1 \leq k < i\}$ and s. Having $\mathbf{I}_M^i$ generated, Algorithm 19 replaces the current content of the set $\mathbf{I}_M$ with the result of the composite operation $\phi(\psi(\mathbf{I}_M, \mathbf{I}_M^i))$. This procedure continues until all tuple-IDs $i \in ID_R$ have been processed. At the end and based on Lemma 4.3, the set $\mathbf{I}_M$ contains all maximum INDs between R and S. At the beginning, we initialize $\mathbf{I}_M$ with $\{\mathbf{I}_{r\rightarrow s}\}$ because $\{\{\mathbf{I}_{r\rightarrow s}\}\}$ is an upper bound of $\mathbf{I}_M$.

The third component of MIND$_2$ is Algorithm 20 called by Algorithm 19 to compute the sets $\mathbf{I}_M^i$ $(1 \leq i \leq |r|)$. It computes them based on Lemma 4.1 and on Lemma 4.2.

Below, we explain these components in details.

4.4.2 Generating unary IND Coordinates

To compute the unary IND coordinates of a $u \in \mathbf{I}_{r\rightarrow s}$, Algorithm 18 opens two cursors at once (Lines 3-6): One for reading the sorted value positions of the attribute $A = LHS(u)$ and the other for reading the sorted value positions of the attribute $B = RHS(u)$ (see Definition 4.2 for the value positions of an attribute). The value positions of every attribute are sorted according to its values in the corresponding relation. In other words, for any $(i_1, v_1), (i_2, v_2) \in P_A$ $(\in P_B)$: The tuple (i_1, v_1) will be read by the corresponding cursor before the tuple (i_2, v_2) if the value v_1 occurs before the value v_2 in the sort sequence. Otherwise, (i_2, v_2) will be read before (i_1, v_1).

In the main *while*-loop (Lines 9-27), Algorithm 18 moves the two cursors in such a way so that it can associate every tuple-ID $i \in ID_R$ with the set of all tuple-IDs $j \in ID_S$ for which both attributes A and B have the same value. In other words, the tuple-ID i is associated with the set

$$\{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\}$$

It saves that association temporary in the hash map *i2jsMap* (Lines 21-23).

After finishing the reading of value positions of P_A and of P_B, Algorithm 18 creates a file for the current unary IND u in the *for*-loop (Lines 1-34) and saves every pair

$$(i, \{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\})$$

in a line in this file. The lines (records) are sorted in ascending order by the left tuple-IDs $i \in ID_R$ and in every line the IDs:

$$j \in \{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\}$$

are also sorted in ascending order (Lines 28-34). This policy of organizing the value positions is required by Algorithm 19.

Algorithm 18: Generation of unary IND Coordinates

Input : $\mathbf{I}_{r \to s}, Repo$
Output : C_u for each $u \in \mathbf{I}_{r \to s}$

```
1  foreach u ∈ I_{r→s} do
2      i2jsMap ← createMap(Int, Set)
3      A ← LHS(u)
4      B ← RHS(u)
5      Cur_A ← createCursor(A)
6      Cur_B ← createCursor(B)
7      (i, v) ← Cur_A.next()
8      (j, v') ← Cur_B.next()
9      while Cur_A.hasNext() and Cur_B.hasNext() do
10         if v = v' then
11             ID_A ← {}
12             ID_B ← {}
13             (k, w) ← Cur_A.current()
14             while v = w do
15                 ID_A = ID_A ∪ {k}
16                 (k, w) ← Cur_A.next()
17             (k, w) ← Cur_B.current()
18             while v = w do
19                 ID_B = ID_B ∪ {k}
20                 (k, w) ← Cur_B.next()
21             if ID_A ≠ ∅ and ID_B ≠ ∅ then
22                 foreach i ∈ ID_A do
23                     i2jsMap.put(i, ID_B)
24         else if v > v' then
25             (j, v') ← Cur_B.next()
26         else
27             (i, v) ← Cur_A.next()
28     writer ← createWriter(u, Repo)
29     ID_A ← i2jsMap.keys()
30     sort(ID_A)
31     foreach i ∈ ID_A do
32         P_B ← i2jsMap.get(i)
33         sort(ID_B)
34         writer.write(u, i, ID_B)
```

Table 4.3 The output of Algorithm 18 for the set of all valid unary INDs between R and S presented in Table 4.1

$A_1 \subseteq B_1$	$A_2 \subseteq B_2$	$A_3 \subseteq B_3$	$A_4 \subseteq B_4$	$A_5 \subseteq B_5$
1, [1]	1, [1]	1, [1, 2, 3]	1, [1, 2, 3]	1, [3]
2, [4, 5]	2, [4, 5]	2, [4]	2, [5]	2, [5]

MIND$_2$ needs only $2 \times |\mathbf{I}_{r \to s}|$ database accesses because every cursor needs a simple *SQL select* statement with an *order by* clause for reading the value positions of an attribute.

Example 4.5. *Based on the attribute value positions listed in Table 4.2, Figure 4.3 illustrates the output of Algorithm 18. Each row in that figure represents a file containing the coordinates of an unary IND.*

4.4.3 Computing maximum INDs between R and S

Algorithm 19, as implementation of Principle 3 (see Section 4.3.3), generates the set of all maximum INDs $\mathbf{I}_M$ by computing the ρ-operator (Definition 4.6) incrementally. It opens all files of the unary INDs coordinates generated by Algorithm 18 and reads them at once (Lines 2-4). Each $u \in \mathbf{I}_{r \to s}$ is associated with a sequential file reader for reading its coordinates C_u. The file readers are managed by a priority queue. For any two readers fr, fr', reader fr has a higher priority than fr' in the queue if and only if the tuple-ID i in the file entry (u,i,L) is smaller than the tuple-ID i' in (u',i',L') where (u,i,L) is the entry that fr can currently read and (u',i',L') is the entry that fr' can currently read. Managing the readers in this way allows Algorithm 19 to collect all unary INDs $u \in \mathbf{I}_{r \to s}$ that have the same tuple-ID i ($i \in ID_R$) in their coordinates (Lines 7-20).

In every pass through the main *while*-loop (Lines 6-32) the algorithm collects the elements (u,L) in the set $\mathbf{L}$ where all unary INDs u in these elements have the same tuple-ID $i \in ID_R$. Each list L in (u,L) is (based on its construction by Algorithm 18) the list of all tuple-IDs $j \in ID_S$, where the values of attribute $RHS(u)$ in these tuples and the value of $LHS(u)$ in tuple i are identical.

After creating the set $\mathbf{L}$ in the current pass of the main *while*-loop for a certain i, Algorithm 19 calls Algorithm 20 to compute the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (Line 21). We denote this set with $\mathbf{I}_M^*$ where the symbol $*$ is a placeholder for any $i \in ID_R$.

After computing the maximum INDs $\mathbf{I}_M^*$ between $\sigma_{ID_R=i}(R)$ and S, the set of all maximum INDs $\mathbf{I}_M$ will be updated by applying the composite operation $\phi(\psi(\mathbf{I}_M, \mathbf{I}_M^*))$ in Line 22 (see Definition 4.4 for ϕ-operator and Definition 4.5 for ψ-operator). The set $\mathbf{I}_M$ is initialized with the set $\{\mathbf{I}_{r \to s}\}$ (Line 5). If the updated set $\mathbf{I}_M$ contains only the unary INDs, the algorithm breaks the main *while*-loop and returns the set of all unary INDs as the maximum INDs (Line 23-27). Otherwise, Algorithm 19 will

Algorithm 19: Generation of maximum INDs

Input : $\mathbf{I}_{r\to s}, Repo$
Output : $\mathbf{I}_M$

1. $Queue \leftarrow$ `createPriorityQueue()`
2. **foreach** $u \in \mathbf{I}_{r\to s}$ **do**
3. $\quad fr \leftarrow$ `createFileReader(`$u, Repo$`)`
4. $\quad Queue$`.add(`fr`)`
5. $\mathbf{I}_M \leftarrow \{\mathbf{I}_{r\to s}\}$
6. **while** $Queue \neq \emptyset$ **do**
7. $\quad \mathbf{L} \leftarrow \emptyset$
8. $\quad Readers \leftarrow \emptyset$
9. $\quad fr \leftarrow Queue$`.pull()`
10. $\quad Readers \leftarrow Readers \cup \{fr\}$
11. $\quad (u,i,L) \leftarrow fr$`.current()`
12. $\quad \mathbf{L} \leftarrow \mathbf{L} \cup \{(u,L)\}$
13. $\quad$ **while** $Queue \neq \emptyset$ **do**
14. $\quad\quad fr' \leftarrow Queue$`.peek()`
15. $\quad\quad (u',i',L') \leftarrow fr'$`.current()`
16. $\quad\quad$ **if** $i \neq i'$ **then break**
17. $\quad\quad fr \leftarrow Queue$`.pull()`
18. $\quad\quad Readers \leftarrow Readers \cup \{fr\}$
19. $\quad\quad (u,i,L) \leftarrow fr$`.current()`
20. $\quad\quad \mathbf{L} \leftarrow \mathbf{L} \cup \{(u,L)\}$
21. $\quad \mathbf{I}_M^* \leftarrow$ `genSubMaxINDs(`$\mathbf{L}, \mathbf{I}_M$`)`
22. $\quad \mathbf{I}_M \leftarrow \phi(\psi(\mathbf{I}_M, \mathbf{I}_M^*))$
23. $\quad$ **foreach** $u \in \mathbf{I}_{r\to s} : \{u\} \in \mathbf{I}_M$ **do**
24. $\quad\quad \mathbf{I}_M \leftarrow \mathbf{I}_M \setminus \{\{u\}\}$
25. $\quad$ **if** $\mathbf{I}_M = \emptyset$ **then**
26. $\quad\quad \mathbf{I}_M \leftarrow \mathbf{I}_{r\to s}$
27. $\quad\quad$ **break**
28. $\quad activeU \leftarrow \cup_{M \in \mathbf{I}_M} M$
29. $\quad$ **foreach** $fr \in Readers$ **do**
30. $\quad\quad$ **if** fr`.hasNext()` *and* $fr.u \in activeU$ **then**
31. $\quad\quad\quad fr$`.next()`
32. $\quad\quad\quad Queue$`.add(`fr`)`

Algorithm 20: Generation of maximum INDs over $\sigma_{ID_R=i}(R)$ and S

```
   Input  : L, I_M^{*-1}
   Output : I_M^*
1  Queue ← createPriorityQueue()
2  foreach (u,L) ∈ L do
3      lr ← createListReader(u, L)
4      Queue.add(lr)
5  UB ← ∅
6  while Queue ≠ ∅ do
7      Readers ← ∅
8      lr ← Queue.pull()
9      Readers ← Readers ∪ {lr}
10     (u, j) ← lr.current()
11     M^{*j} ← {u}
12     while Queue ≠ ∅ do
13         lr' ← Queue.peek()
14         (u', j') ← lr'.current()
15         if j ≠ j' then break
16         lr ← Queue.pull()
17         Readers ← Readers ∪ {lr}
18         (u, j) ← lr.current()
19         M^{*j} ← M^{*j} ∪ {u}
20     if ∃M ∈ I_M^{*-1} : M ⊆ M^{*j} then
21         UB ← UB ∪ {M}
22     if UB = I_M^{*-1} then
23         I_M^* ← I_M^{*-1}
24         break
25     I_M^* ← I_M^* ∪ {M^{*j}}
26     foreach lr ∈ Readers do
27         if lr.hasNext() then
28             lr.next(); Queue.add(lr)
29 I_M^* ← φ(I_M^*)
```

update the queue only with readers of those unary INDs u which are contained in at least one set of $\mathbf{I}_M$ (Lines 28-32).

4.4.4 Computing maximum INDs between $\sigma_{ID_R=i}(R)$ and S

Based on Principle 1 and Principle 2, Algorithm 20 computes the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S from the set $\mathbf{L}$ while it exploits the set $\mathbf{I}_M^{*-1}$ to improve the performance. The set $\mathbf{L}$ generated by Algorithm 19 (Lines 7-20) contains the elements (u,L): all unary INDs in these elements have the same left tuple-ID i in their coordinates while every list L in (u,L) is the sorted list of all tuple-IDs $j \in ID_S$ in the coordinates $(i,j) \in C_u$. The algorithm reads all the lists in the set $\mathbf{L}$ at once and uses a priority queue to manage the list readers in the same way in which Algorithm 19 manages the file readers of the unary INDs coordinates.

In the main *while*-loop, we collect all unary INDs u in the set M^{*j} that have the same tuple-ID j in their coordinates (Lines 7-19). The symbol $*$ in M^{*j} is a placeholder for the corresponding i. Thus, based on the properties of the elements (u,L) of the set $\mathbf{L}$, the set M^{*j} contains all unary INDs u that have (i,j) in their coordinates C_u. That means that, according to Lemma 4.1, M^{*j} is the maximum IND between $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$.

Every computed set M^{*j} is added to the set $\mathbf{I}_M^*$ (Line 25). Updating $\mathbf{I}_M^*$ by applying the ϕ-operator on it gives us, according to Lemma 4.2, the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (Line 26).

The objective of the input set $\mathbf{I}_M^{*-1}$ is to improve the performance of computing $\mathbf{I}_M^*$. The set $\mathbf{I}_M^{*-1}$ is the set of all maximum INDs between $\sigma_{ID_R<i}(R)$ and S. For every generated set M^{*j}, Algorithm 20 checks if there is a set M in $\mathbf{I}_M^{*-1}$ such that M is a subset of M^{*j} (Lines 20-21). If such a set exists, it is added to the set UB. If the set UB contains all sets from $\mathbf{I}_M^{*-1}$, then the algorithm breaks the execution, and returns $\mathbf{I}_M^{*-1}$ as the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (Lines 22-24). This rule does not have any effect on the correctness of Algorithm 19 because the result of the composite operation $\phi(\psi(\mathbf{I}_M,\mathbf{I}_M^*))$ in Algorithm 19 is the set $\mathbf{I}_M$ itself if every set in $\mathbf{I}_M^*$ is a superset of a set in $\mathbf{I}_M$.

4.5 Experimental evaluation

The main aim of our experiments in this chapter is to compare the scalability of MIND$_2$ with that of FIND$_2$. That is our focus because FIND$_2$ is developed to reduce the number of IND candidates required by the approach of MIND. Although ZIGZAG is also designed to handle long INDs, we limited our experiments to FIND$_2$ because, as discussed in Sections 4.1 and 4.6, FIND$_2$ and ZIGZAG approach the n-ary IND discovery problem from similar directions and have many properties in common.

4.5.1 Setup

Datasets The conducted experiments are divided into three groups, where each group has its own dataset described in the corresponding paragraph of the following subsection. In general, the number

of rows varies between 500,000 and 16,000,000 rows. The other important variable that has a big impact on the scalability of discovering the n-ray INDs between two relations is the number of valid unary INDs. The number of valid unary INDs in these experiments varies between 8 and 19 unary INDs over the corresponding table pairs.

Experimental conditions We performed the experiments on Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We used Oracle 11g as the database server installed on the same machine. We implemented both algorithms in 64-bit Java 7. We implemented FIND$_2$ based on Koeller and Rundensteiner [2002]. For MIND$_2$, we set the minimum Java heap size to 4 GB and the maximum to 6 GB. While for FIND$_2$, we set the Java stack size to 4 GB. FIND$_2$ validates IND candidates by applying the *SQL* query proposed in DeMarchi et al. [2009].

4.5.2 Evaluation of the performance

Experiments of Group 1 The purpose of these experiments is to compare the scalability of MIND$_2$ with that of FIND$_2$ using the real-word dataset MUSICBRAINZ available at https://musicbrainz.org. MUSICBRAINZ is an open music encyclopedia that collects music metadata and makes them available to the public. MUSICBRAINZ contains 27 GB of data. It contains 178 non-empty tables (relations) with 1053 non-empty columns (attributes). The dataset has a total of 44,803 valid unary INDs. We detected pairs of tables where there is at least one valid n-ary IND with size greater than 2 between the tables of each pair. The number of tuples varies between 500,000 and 1,000,500 tuples. The results of these experiments are presented in Table 4.4. The acronym „TP." stands for table pair. The left part of Table 4.4 shows some statistics about detected INDs: the number of valid unary INDs ($|\mathbf{I}_{r\rightarrow s}|$), the number of detected maximum INDs ($|\mathbf{I}_M|$), the size of the longest maximum INDs (n_{max}) accompanied by their number ((x Nr.)), and the size of shortest maximum INDs (n_{min}) accompanied by their number ((x Nr.)).

The right part of Table 4.4 shows the needed time (in minutes) by MIND$_2$ and FIND$_2$ for detecting the valid INDs, respectively. The acronym „o.o.M." refers to out of memory exception. In most of these experiments, MIND$_2$ outperforms FIND$_2$ significantly. Furthermore, they show that MIND$_2$'s scalability, on the contrary to that of FIND$_2$, is robust and not sensitive to the high number of small valid INDs. The reason why FIND$_2$ terminates with an *out of memory* exception is the complexity of hypergraphs created by FIND$_2$. If one of these hypergraphs is not sparse (irreducible), then the hyperclique-finding subroutine presented in Koeller and Rundensteiner [2002] attempts to simplify the corresponding hypergraph by removing hyperedges from it. The removing of hyperedges performed by this subroutine of FIND$_2$ is not defined deterministically. This behavior causes a lot of recursive calls and consumes a huge amount of memory. FIND$_2$ needed less time than MIND$_2$ only for the table pair 5 and 7, respectively. This is because the created hypergraphs for these table pairs are sparse, respectively.

Table 4.4 Comparing MIND$_2$'s runtime with FIND$_2$'s runtime using MUSICBRAINZ database (o.o.M. = out of memory, m = minutes)

TP.	$\lvert \mathbf{I}_{r\to s} \rvert$	$\lvert \mathbf{I}_M \rvert$	n_{max} (x Nr.)	n_{min} (x Nr.)
1	19	75	5 (x 2)	2 (x 4)
2	17	25	3 (x 13)	2 (x 12)
3	15	28	3 (x 17)	2 (x 11)
4	15	56	3 (x 56)	-
5	14	28	3 (x 20)	2 (x 8)
6	13	23	3 (x 6)	2 (x 17)
7	12	26	3 (x 19)	2 (x 7)
8	12	11	3 (x 11)	-

TP.	MIND$_2$	FIND$_2$
1	184 m	o.o.M. after 250 m
2	4 m	o.o.M. after 40 m
3	2 m	o.o.M. after 33 m
4	1.5 m	o.o.M. after 322 m
5	15 m	4 m
6	15 m	o.o.M. after 33 m
7	22 m	6 m
8	11 m	30 m

Experiments of Group 2 The purpose of these experiments is to compare MIND$_2$'s performance with the performance of the best case for FIND$_2$. The best case for FIND$_2$ is when FIND$_2$ needs to build only the 2-hypergraph and then finds only one clique representing a valid IND. This happens for example when the database contains only one valid IND σ of size $n > 2$. In this case, FIND$_2$ needs $n \times (n-1)/2$ database access to enumerate the valid binary INDs and one access to validate the clique.

To demonstrate this case, we generated two synthetic databases DB 1 and DB 2. Both databases contain 16,000,000 tuples. DB 1 contains one valid maximum IND in size 9, while DB 2 contains one valid maximum IND in size 10. The results of these experiments are presented in Table 4.5 (rows 1 and 2 in each part of Table 4.5). FIND$_2$'s runtime is dominated by the runtime of the required *SQL* queries for enumerating the valid binary INDs. Therefore, MIND$_2$ is up to 8x faster than FIND$_2$.

Experiments of Group 3 The purpose of these experiments is to show that in some cases FIND$_2$ needs the same exponential number of database accesses as needed by MIND's approach. Let $\sigma = \{u_1, \ldots, u_n\}$ be an invalid n-ary IND with the property that every $(n-1)$-ary IND contained in σ is a valid IND. In this case, FIND$_2$ builds $n-2$ k-hypergraphs $(2 \leq k \leq n-1)$ where every k-hypergraph has $\binom{n}{k}$ edges and contains only the same clique, namely $\{u_1, \ldots, u_n\}$. Thus, FIND$_2$ needs $\binom{n}{2} + \cdots + \binom{n}{n-1} + (n-1) = 2^n - 3$ *SQL* queries to discover the n valid $(n-1)$-ary INDs contained in σ.

To illustrate this case, we also generated two synthetic databases DB 3 and DB 4, where every database has 10,000,000 tuples in average. DB 3 contains 8 valid INDs in size 8, while DB 4 contains 9 valid INDs in size 9. Table 4.5 (rows 3 and 4 in each part of this table) presents the results of these experiments. The FIND$_2$'s runtime is dominated by the exponential number of the database accesses

Table 4.5 Results of the experiments in groups 2 and 3 (# = number of, m = minutes)

DB	$\|\mathbf{I}_M\|$	n_{max} (x Nr.)
1	1	9 (x 1)
2	1	10 (x 1)
3	8	8 (x 8)
4	9	9 (x 9)

	#DB-Accesses		Runtime	
DB	FIND$_2$	MIND$_2$	FIND$_2$	MIND$_2$
1	37	18	57 m	11 m
2	46	20	100 m	12 m
3	509	18	263 m	9.5 m
4	1021	20	906 m	11.5 m

needed for the validation of the IND candidates. Therefore, MIND$_2$ is much more (up to 82x) faster than FIND$_2$.

4.6 Related work

DeMarchi et al. [2002, 2009] presented MIND that applies the levelwise approach to generate IND candidates (see Subsection 1.8.1). MIND generates all $(k+1)$-IND candidates from the valid k-INDs and the valid unary INDs. It is based on the view that the validity of $\sigma_1 = R[A_1,\ldots,A_k] \subseteq S[B_1,\ldots,B_k]$ and $\sigma_2 = R[A_{k+1}] \subseteq S[B_{k+1}]$ is a necessary but not sufficient condition for $\sigma = R[A_1,\ldots,A_k,A_{k+1}] \subseteq S[B_1,\ldots,B_k,B_{k+1}]$ to be valid. So, if σ_1 or σ_2 is invalid, then it is impossible for σ to be valid. In this case, σ is pruned and no testing for its validity is necessary. In the other case, if both σ_1 and σ_2 are valid, then σ has a chance to be valid and therefore becomes a candidate of size $k+1$. This candidate is then validated against the database. After Generating and testing all $(k+1)$-ary IND candidates, MIND generates and tests the $(k+2)$-ary IND candidates.

DeMarchi and Petit [2003] developed the ZIGZAG algorithm based on borders of theories developed by Mannila and Toivonen [1997]. Initially and for a k specified by the user, ZIGZAG initializes the positive border and the negative border by applying an adaptation of the levelwise algorithm MIND until the level k is reached. Furthermore, ZIGZAG introduces the optimistic positive border computed by finding the minimal hypergraph traversals in the hypergraph generated from the negative border. The algorithm iteratively updates the three borders as long as the optimistic positive border contains INDs that are not contained in the positive border. Every updating process combines a pessimistic bottom-up search with an optimistic top-down search. In the bottom-up search, ZIGZAG validates the IND candidates against the database. In the top-down search, ZIGZAG estimates the distance between the invalid INDs and the positive border by counting the number of tuples that do not satisfy these invalid INDs.

Koeller and Rundensteiner [2002, 2003] proposed FIND$_2$ that begins by exhaustively validating the unary and binary INDs, thereby forming a 2-uniform hypergraph using the valid unary INDs as nodes and the valid binary INDs as edges. Then, FIND$_2$ proceeds in stages enumerated by $k = 2, 3, \ldots$. In each stage k, all hypercliques in the k-hypergraph are identified by HYPERCLIQUE algorithm [Koeller and Rundensteiner, 2002], where every hyperclique represents an IND candidate. Next, IND candidates are checked for validity in the database. Each invalid IND corresponding to a hyperclique in the k-hypergraph is broken into all $(k+1)$-ary INDs contained in it. Afterward, the $(k+1)$-ary INDs form the edges of the $(k+1)$-hypergraph. Edges corresponding to the invalid $(k+1)$-ary INDs are removed from the $(k+1)$-hypergraph. The process is repeated for increasing k until no new cliques are found.

A common problem with the principle of candidate generation, applied by MIND, ZIGZAG, and FIND$_2$, is the poor scalability in the length of the longest valid IND in the dataset as the validation of an n-ary IND (n > 1) may first require checking all (exponentially many in n) INDs implied by it.

To this end, we devised MIND$_2$ [Shaabani and Meinel, 2016], the first algorithm to discover n-ary INDs without candidate generation.

4.7 Conclusion and future work

In this chapter, we developed novel characterizations of the maximum inclusion dependencies. We achieved these characterizations by defining operations on the unary IND coordinates–it is also a new concept introduced in this chapter. Applying these operations on the unary IND coordinates enables the inference of the maximum inclusion dependencies without any candidate generation–this has a big impact on the scalability of discovering long n-ary INDs..

One possible implementation of the developed principles for detecting the maximum INDs is MIND$_2$, which has been presented in this chapter. MIND$_2$ generates the unary IND coordinates by accessing the database only 2 $\times$ the number of valid uINDs. This means that MIND$_2$ has eliminated the exponential number of database accesses needed by the other algorithms. The experimental evaluations showed that MIND$_2$ outperforms FIND$_2$ significantly and that MIND$_2$'s scalability is not influenced by a high number of small valid INDs in contrast to FIND$_2$'s scalability.

The work is the main milestone for further research on how to develop a distributed version of MIND$_2$ to parallelize both the generation of the unary IND coordinates and the computation of the maximum INDs. A distributed implementation has to reduce the dependence of MIND$_2$'s performance on the number of tuples.

Chapter 5

Conclusion

Inclusion dependencies express redundancies between datasets. Such redundancies are increasingly common, as more data is produced, gathered, and stored by many applications. Any effort to combine or link such data must rely on knowledge of inclusion dependencies (INDs).

In this thesis, we developed three novel algorithms for discovering INDs: S-INDD++ for discovering unary INDs (uINDs), a system for updating uINDs in dynamic data, and MIND_2 for discovering the maximum INDs without candidate generation.

S-INDD++ [Shaabani and Meinel, 2018b] significantly outperforms existing algorithms for discovering uINDs by eliminating their shortcomings. In fact, experiments conducted on large datasets with thousands of attributes and more than 200 million tuples showed that S-INDD++ reduces the runtime of the state-of-the-art algorithm [Papenbrock et al., 2015] by up 50 %.

S-INDD++ is based on a new attribute clustering concept from which all uINDs are efficiently derivable [Shaabani and Meinel, 2015]. The efficiency of this inference is articulated in the elimination of all redundant intersection operations caused by the generation of uINDs from an inverted index applied by other algorithms in the related work. The experimental evaluation of the attribute clustering showed that the reduction in the intersection operations is up to 99.999 %.

S-INDD++ applies a new partitioning strategy that divides the dataset into disjoint subsets in different sizes to enable pruning a large number of useless attributes in early phases of the detection process by processing the first partitions of smaller sizes. In this regard, useless attributes are those attributes that are not a part of any IND in the given dataset. Based on the attribute clustering of a partition, S-INDD++ determines which attributes to be excluded from processing the subsequent partitions. S-INDD++ computes the attribute clustering of the entire dataset as the union of all clusters of all partitions. When a partition contains values of an attribute, they are divided into sorted subsets stored in a certain format in the repository representing the partition. If an attribute is not discarded by the processing of a previous partition, S-INDD++ completes sorting its values in the partition currently processed using an adaption of the external merge-sort algorithm; otherwise, S-INDD++ ignores the values. In this way, S-INDD++ reduces the time needed for sorting the values of the useless attributes. Having the value subset of each active attribute sorted in the current partition is

required for generating the attribute clustering of that partition. S-INDD++ generates the attribute clustering of each partition by calling S-INDD.

S-INDD [Shaabani and Meinel, 2015] has solved the scalability problem of SPIDER [Bauckmann et al., 2006]. S-INDD can be configured to present SPIDER as a special case of it. S-INDD is a composite of configurable computing iterations, in each of which S-INDD controls the number of attributes that have to be aggregated to generate the clusters gradually. This flexibility makes S-INDD scale well when the number of attributes increases considerably. Experiments showed that by increasing the number of attributes from 6000 to 7000, the runtime of SPIDER increases by 38 %, while the runtime of S-INDD increases by only one percent.

Based on the attribute clustering, the first approach for incrementally updating uINDs in frequently changing datasets has been developed in the thesis [Shaabani and Meinel, 2017, 2018a]. In this regard, the attribute clustering is augmented with new operators applied to the clusters after each update of the dataset. This means that the incremental update of uINDs is reduced to the problem of incrementally updating the attribute clustering. The justification for this reduction is the efficiency of inferring the uINDs from the clusters after an update of the data. Indexed data structures, caching strategies, and algorithms are designed for an efficient implementation of the cluster operators. A comprehensive evaluation conducted on large datasets with hundreds of attributes and more than one hundred million tuples has shown that the incremental update reduces the runtime of a complete rediscovery by up to 99.9996 %.

To initialize the data structures when the incremental discovery (i.e., the incremental update) has to start with a large legacy dataset, an extended version of S-INDD is developed. A sharing-nothing architecture is designed to scale out the incremental discovery of unary inclusion dependencies. Moreover, an approach to incrementally discovering approximate uINDs is suggested.

All existing algorithms for discovering n-ary INDs are based on the principle of candidate generation–a common problem with this principle is the poor scalability in the size of the largest valid inclusion dependency since validation of an IND of a size n greater than one might require checking all INDs (exponentially many in n) implied by it. To solve this problem, the thesis developed MIND_2 [Shaabani and Meinel, 2016], the first algorithm for discovering n-ary INDs without candidate generation. MIND_2 is an implementation of a novel mathematical framework developed in the thesis for inferring the maximum inclusion dependencies from which all other n-ary INDs are derivable. The framework characterizes the maximum inclusion dependencies as the result of operations defined on the uIND coordinates–a new concept introduced in the thesis. For generating the uINDs coordinates, MIND_2 needs only $2 \times$ the number of valid uINDs database accesses. The experimental evaluation conducted on real and synthetic dataset has showed that MIND_2 is more scalable and effective than graph-based algorithms like FIND_2.

The performance of exhaustively discovering n-ary INDs suffers from dimensionality curse–it depends not only on the number of valid unary INDs, but, for the most part, also on the number of tuples. MIND_2 eliminates the exponential number of database accesses, but its performance is bounded by the quadratic number of tuples. Therefore, there is still a need for further research on

how to extend and implement our framework for the inference of the maximum INDs in such a way that scales in the two dimensions of the problem–the number of tuples and the size of the largest maximum IND. Since MIND$_2$ is decoupled from accessing the database after the generation of unary IND coordinates, one promising direction is to develop a distributed execution of MIND$_2$ to scale the computation in the number of tuples.

Bibliography

Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Detecting unique column combinations on dynamic data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1036–1047, 2014.

Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4):557–581, August 2015. ISSN 1066-8888.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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, Sam Madden, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, Kenneth Ross, Cyrus Shahabi, Dan Suciu, Shiv Vaithyanathan, and Jennifer Widom. Challenges and opportunities with big data: A white paper prepared for the computing community consortium committee of the computing research association. Technical report, 2012. URL http://cra.org/ccc/resources/ccc-led-whitepapers/. Accessed on 19.10.2017.

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499, 1994.

Jana Bauckmann. *Dependency Discovery for Data Integration*. PhD thesis, Hasso Plattner Institute at the University of Potsdam, 2013. URL http://opus.kobv.de/ubp/volltexte/2013/6664/. Accessed on 9.01.2019.

Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently computing inclusion dependencies for schema discovery. In *22nd International Conference on Data Engineering Workshoops (ICDEW'06)*, pages 2–2, 2006.

Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. Discovering conditional inclusion dependencies. In *CIKM*, pages 2094–2098. ACM, 2012.

Siegfried Bell and Peter Brockhausen. Discovery of data dependencies in relational databases. Technical report, Universität Dortmund, 1995.

Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. The parameterized complexity of dependency detection in relational databases. In *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, pages 6:1–6:13, 2016.

Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 746–755, 2007.

Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 243–254, 2007.

Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. Relaxed functional dependencies - A survey of approaches. *IEEE Trans. Knowl. Data Eng.*, 28(1):147–165, 2016.

Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

Marco A. Casanova, Luiz Tucherman, and Antonio L. Furtado. Enforcing inclusion dependencies and referencial integrity. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 38–49, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc. ISBN 0-934613-75-3.

Jianer Chen and Fenghui Zhang. On product covering in 3-tier supply chain models: Natural complete problems for $W[3]$ and $W[4]$. *Theortical Computer Science*, 363(3):278–288, 2006.

Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 238–249. Springer, 2006.

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

Edgar Frank Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 315–326, 2007.

Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. All aboard the databus!: Linkedin's scalable consistent change data capture platform. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 18, 2012.

Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 240–251, 2002.

DB-ENGINES. DBMS popularity broken down by database model. Technical report, 2019. URL https://db-engines.com/en/ranking_categories. Accessed on 8.01.2019.

Fabien DeMarchi and Jean-Marc Petit. Zigzag: a new algorithm for mining large inclusion dependencies in database. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), 19-22 December 2003, Melbourne, Florida, USA*, pages 27–34, 2003.

Fabien DeMarchi and Jean-Marc Petit. Approximating a set of approximate inclusion dependencies. In *Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM'05 Conference held in Gdansk, Poland, June 13-16, 2005*, pages 633–640, 2005.

Fabien DeMarchi, Stéphane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. In *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, pages 464–476, 2002.

Fabien DeMarchi, Stéphane Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.*, 32(1):53–73, 2009.

Evoke Software. Data profiling and mapping. The essential first step in data migration and integration projects. Technical report, 2000. URL http://ciains.info/elearning/Solutions/ANew/DataMigrationFirstSteps.pdf. Accessed on 20.08.2018.

Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 159–170, 2008.

Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *PVLDB*, 2(1):407–418, 2009.

Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. Incremental detection of inconsistencies in distributed data. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 318–329, 2012.

Patrick C. Fischer, Lawrence V. Saxton, Stan J. Thomas, and Dirk Van Gucht. Interactions between dependencies and nested relational structures. *Journal of Computer and System Sciences*, 31(3): 343–354, 1985.

Peter A. Flach and Iztok Savnik. Database dependency discovery: A machine learning approach. *AI Commun.*, 12(3):139–160, 1999.

Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theor. Comput. Sci.*, 26:149–195, 1983.

Lukasz Golab, Howard J. Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. Sequential dependencies. *PVLDB*, 2(1):574–585, 2009.

Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. Incremental record linkage. *PVLDB*, 7(9): 697–708, 2014.

Jarek Gryz. Query folding with inclusion dependencies. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 126–133, 1998.

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 157–166, 1993.

Marc Gyssens. Database dependencies. In *Encyclopedia of Database Systems*, pages 704–708. 2009.

Sven Hartmann and Sebastian Link. Multi-valued dependencies in the presence of lists. In *PODS*, pages 330–341. ACM, 2004.

Sven Hartmann and Sebastian Link. On inferences of full hierarchical dependencies. In *ACSC*, volume 62 of *CRPIT*, pages 69–78. Australian Computer Society, 2007.

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD Conference*, pages 647–658. ACM, 2004.

Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *Int. J. Intell. Syst.*, 7(7):591–607, 1992.

Benjamin Kille, Frank Hopfgartner, Torben Brodt, and Tobias Heintz. The plista dataset. In *Proceedings of the 2013 International News Recommender Systems Workshop and Challenge*, NRS '13, pages 16–23, 2013. ISBN 978-1-4503-2302-4.

Jyrki Kivinen and Heikki Mannila. Approximate inference of functional dependencies from relations. *Theor. Comput. Sci.*, 149(1):129–149, 1995.

Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016.

Andreas Koeller. *Integration of Heterogeneous Databases: Discovery of Meta-Information and Maintenance of Schema-Restructuring Views*. PhD thesis, Worcester Polytechnic Institute, MA, USA, 2002. URL https://digitalcommons.wpi.edu/etd-dissertations/116. Accessed on 9.01.2019.

Andreas Koeller and Elke A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. Technical Report WPI-CS-TR-02-15, Worcester Polytechnic Institute, 2002. Accessed on 9.01.2019.

Andreas Koeller and Elke A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 683–685, 2003.

Andreas Koeller and Elke A. Rundensteiner. Heuristic strategies for inclusion dependency discovery. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25-29, 2004, Proceedings, Part II*, pages 891–908, 2004.

Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1275–1278, 2009.

Sebastian Kruse, Anja Jentzsch, Thorsten Papenbrock, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Rdfind: Scalable conditional inclusion dependency discovery in RDF datasets. In *SIGMOD Conference*, pages 953–967. ACM, 2016.

Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. Fast approximate discovery of inclusion dependencies. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pages 207–226, 2017.

Mark Levene and Millist W. Vincent. Justification for inclusion dependency normal form. *IEEE Trans. Knowl. Data Eng.*, 12(2):281–291, 2000.

Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data- a review. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):251–264, 2012. ISSN 1041-4347.

Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systtems*, 27(1):1–19, 2002.

Michael J. Maher. Constrained dependencies. *Theoretical Computer Science*, 173(1):113–149, 1997.

David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258, January 1997. ISSN 1384-5810.

Mozhgan Memari, Sebastian Link, and Gillian Dobbie. SQL data profiling of foreign keys. In *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, pages 229–243, 2015.

Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Ling-Ling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

Felix Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2013.

Sam Newman. *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7.

Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 821–833, 2016.

Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *PVLDB*, 8(7):774–785, 2015.

Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. Technical report, 2017. URL http://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf. Online; accessed on 10.09.2018.

Jan Renz, Gerado Navarro-Suarez, Rowshan Sathi, Thomas Staubitz, and Christoph Meinel. Enabling schema agnostic learning analytics in a service-oriented MOOC platform. In *Proceedings of the Third ACM Conference on Learning @ Scale, L@S 2016, Edinburgh, Scotland, UK, April 25 - 26, 2016*, pages 137–140, 2016.

Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *12th International Workshop on the Web and Databases, WebDB 2009, Providence, Rhode Island, USA, June 28, 2009*, 2009.

Barna Saha and Divesh Srivastava. Data quality: The other face of big data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1294–1297, 2014.

Iztok Savnik and Peter A. Flach. Discovery of multivalued dependencies from relations. *Intell. Data Anal.*, 4(3-4):195–211, 2000.

Ingo Schmitt and Gunter Saake. A comprehensive database schema integration method based on the theory of formal concepts. *Acta Informatica*, 41(7-8):475–524, 2005.

Nuhad Shaabani and Christoph Meinel. Scalable inclusion dependency discovery. In *Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I*, pages 425–440, 2015.

Nuhad Shaabani and Christoph Meinel. Detecting maximum inclusion dependencies without candidate generation. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, pages 118–133, 2016.

Nuhad Shaabani and Christoph Meinel. Incremental discovery of inclusion dependencies. In *Proceedings of the 29th ACM International Conference on Scientific and Statistical Database Management, SSDBM 2017, Chicago, IL, USA, June 27-29, 2017*, pages 2:1–2:12, 2017.

Nuhad Shaabani and Christoph Meinel. Incrementally updating unary inclusion dependencies in dynamic data. *Journal of Distributed and Parallel Databases*, pages 1–44, August 2018a. ISSN 0926-8782. URL https://doi.org/10.1007/s10619-018-7233-5.

Nuhad Shaabani and Christoph Meinel. Improving the efficiency of inclusion dependency detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 207–216, 2018b.

Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry C. Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 351–366, 2015.

Kenneth P. Smith, Leonard J. Seligman, Arnon Rosenthal, Chris Kurcz, Mary Greer, Catherine Macheret, Michael Sexton, and Adric Eckstein. "big metadata": The need for principled metadata management in big data ecosystems. In *Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference*, pages 13:1–13:4, 2014.

Shaoxu Song and Lei Chen. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3):16:1–16:41, 2011.

Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.

Pauray S. M. Tsai, Chih-Chong Lee, and Arbee L. P. Chen. An efficient approach for incremental association rule mining. In Ning Zhong and Lizhu Zhou, editors, *Methodologies for Knowledge Discovery and Data Mining*, pages 74–83, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48912-2.

Shyue-Liang Wang, Wen-Chieh Tsou, Jiann-Horng Lin, and Tzung-Pei Hong. *Maintenance of Discovered Functional Dependencies: Incremental Deletion*, pages 579–588. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-44999-7.

Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219, Apr 2008. ISSN 1573-756X.

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.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Doktorarbeit mit dem Titel:

On Discovering and Incrementally Updating Inclusion Dependencies

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ferner wurde die vorliegende Arbeit an keiner anderen Hochschule eingereicht.

Potsdam, den 30. Januar 2019

Nuhad Shaabani