Universiteit Antwerpen

Faculteit Wetenschappen

Informatica

Mining Patterns in Relational Databases

Proefschrift voorgelegd tot het behalen van de graad van doctor in de wetenschappen: informatica aan de Universiteit Antwerpen, te verdedigen door

Wim Le Page

Promotor: Prof. dr. Bart Goethals

Antwerpen, 2009

Mining Patterns in Relational Databases

Wim Le Page
Universiteit Antwerpen, 2009

Universiteit
Antwerpen

http://www.universiteitantwerpen.be

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted, provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice. Copyrights for components of this work owned by others than the author must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission of the author.

BELGIAN SCIENCE POLICY

Part of this work was carried out in the context of the Interuniversity Attraction Poles Programme – Belgian State – Belgian Science Policy, project *MoVES*.

Typesetting by LaTeX

Acknowledgements

This dissertation would not have been possible without the support, guidance and collaboration of many people who I would like to acknowledge. First I would like to thank Jan Paredaens and Bart Goethals who helped me start this PhD in the first place. As promotor, Bart has always provided valuable ideas and feedback to advance my research, but he has also been a colleague without whom many late night deadlines and conference trips would not have been the same. I also appreciate the help of the other people who were closely involved with the research presented in this dissertation. I would like to thank Heikki Mannila for his comments and suggestions shared during my short but pleasant stay in Helsinki. Many thanks go out to Dominique Laurent. This dissertation would just not have been the same without the many conversations, real as well as e-mail, we had. I am also very grateful to Michael Mampaey, who has not only been a long time friend, but also an invaluable collaborator on the research performed for this dissertation.

I would also like to thank all of my colleagues at the University of Antwerp for providing a pleasant and productive atmosphere. Thank you to all the current and some of the past members of the ADReM team: Adriana, Alvaro, Bart, Boris, Calin, Jan, Jan, Jeroen, Jilles, Koen, Michael, Nele, Nikolaj, Philippe, Roel and Toon. The unique atmosphere during our meetings and excursions and the many interesting discussions about language and culture made working in this team a very enjoyable experience. Special thanks go out to colleagues Jeroen and Olaf for spending time with me contemplating and (publicly) sharing PhD experiences. Many thanks to Joris who helped me in ‘getting things done’ as well as with many other things. In addition I would like to thank Juan who together with Jeroen, Joris and Michael made many of the work breaks very playful. I also enjoyed the company and smalltalk of all the colleagues from other universities I met at the conferences I attended.

Finally I would like to thank my family and friends for their support. Thanks to Johan, Michael and Wesley for providing much needed diversion. Special thanks to my parents Alice and Wilfried and my sister Els for always believing in me and helping me through the tougher periods. Most of all I would like to thank my wife, Debby, for her unconditional support, understanding and motivation.

Wilrijk, December 2009

Abstract

The Information Age has provided us with huge data repositories which cannot longer be analysed manually. The potential high business value of the knowledge that can be gained, drives the research for automated analysis methods that can handle large amounts of data. Most of the data of industry, education and government is stored in relational database management systems (RDBMSs). This motivates the need for data mining algorithms that can work with arbitrary relational databases, without the need for manual transformation and preprocessing of the data. In this dissertation we introduce two data mining approaches towards the goal of mining interesting patterns in arbitrary relational databases.

We first examine *frequent query mining*. Within this setting we propose a novel algorithm Conqueror, that is capable of efficiently generating and pruning the search space of all *simple conjunctive queries*, a pattern class capable of expressing many different kinds of interesting patterns, such as, but not limited to, functional and inclusion dependencies. We present experiments, showing the feasibility of our approach, as well as the expressiveness of simple conjunctive queries.

Using the knowledge that our algorithm is capable of detecting functional dependencies and foreign keys that were not given initially, we extend our algorithm, enabling it to *use* these newly discovered, previously unknown functional dependencies and foreign keys to produce a *concise* set of interesting patterns, free of redundant information. We implement and test our updated algorithm, Conqueror$^+$, and show that it efficiently reduces the number of queries generated. Furthermore we experimentally show that this reduction has the added benefit that Conqueror$^+$ also outperforms Conqueror in timing.

As a second approach we investigate *frequent relational itemset mining*. We introduce a novel efficient propagation-based depth-first algorithm, called SMuRFIG that is capable of mining *frequent relational itemsets* as well as confident *relational* *association rules.* We introduce a new support measure and show its usefulness in expressing interesting patterns. In addition we define the deviation measure to address the statistical pattern blow-up intrinsic to the relational case, and show that it works as promised. We theoretically and experimentally show that the SMuRFIG algorithm is scalable with respect to time and memory. Moreover, we generalise some popular redundancy measures – closure-based redundancy and minimal improvement – to the multi-relational setting, and confirm that they can reduce the output when complete results are not desired.

This dissertation shows that both approaches provide us with practical algorithms towards the ultimate goal of discovering patterns in arbitrary relational databases.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

DATA generation and collection capabilities have increased rapidly over the past decades. The increased digitisation and automation of business, education, government and individuals as well as the increased collection of consumer data in various industries and the ever increasing amount of information available through the Internet are only a few factors contributing to this trend. This explosive growth of information available to us, also creates the urgent need for techniques that can analyse this data. Due to the sheer volume of information, a completely manual process is no longer achievable. Automation trough the use of intelligent algorithms is therefore a key aspect of data analysis in this 'Information Age'. *Data mining*, also referred to as *knowledge discovery from data (KDD)*, is 'the automated or convenient extraction of patterns representing knowledge implicitly stored or captured in large databases, data warehouses, the Web, other massive information repositories, or data streams' [Han & Kamber, 2006]. Data mining relates to many disciplines including database technology, machine learning, statistics, information retrieval, pattern recognition and data visualisation. In this chapter we will first try to define what data mining is. Then we give an introduction to *relational databases*, currently the most widely used database type to store the wealth of information currently available to us. Finally we will provide a short overview of the chapters of this dissertation explaining our approach to *mining patterns in relational databases.*

1.1 Data Mining

The term *data mining* refers to the extraction or ‘mining’ of valuable knowledge from large amounts of data, in analogy to industrial mining where small sets of valuable nuggets (e.g. gold) are extracted from a great deal of raw material (e.g. rocks). Data mining is part of the knowledge discovery process depicted in Figure 1.1. In this process, selection and transformation are forms of preprocessing, where one selects part of the complete *database* and possibly transforms it into a certain form required for the next step. Often data cleaning and data integration are also part of this initial phase of data preparation. The resulting *data* is the input for the data mining phase, which in its turn results in discovered *patterns*. The interesting patterns are then presented to the user. As these patterns can be stored as new knowledge in a knowledge base, they can, in turn, again be considered as input for another knowledge discovery process.

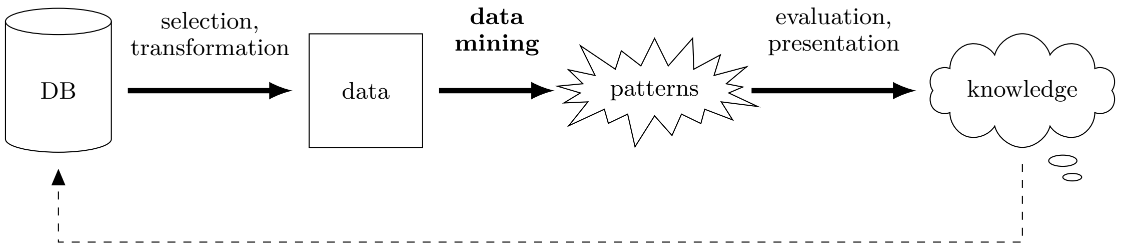


Figure 1.1: Data Mining as part of the Knowledge Discovery Process

A typical example of a knowledge discovery process is market basket analysis. Here, the database consists of the purchase-records of a supermarket. From this database we select the distinct transactions or ‘baskets’ bought by customers and transform these into sets of products. Then the data mining step performs *frequent itemset mining*, where sets of products that are frequently bought together are discovered. Finally these frequent sets are presented to the analyst, who can then use this new knowledge for marketing or other purposes. A more in-depth introduction to frequent itemset mining is provided in Chapter 2.

It is clear that data mining is a crucial part of the knowledge discovery process. Using data mining, interesting knowledge, recurring patterns or high-level information can be extracted from a database and then analysed further. This knowledge can be used in decision making, optimisation of processes and information management. Data mining is therefore considered ‘one of the most important frontiers in database and information systems and one of the most promising interdisciplinary developments in the information technology’ [Han & Kamber, 2006].

The field of data mining can be classified along several axes. One possible distinction that can be made is that between descriptive and predictive data mining. In *descriptive* data mining the goal is to characterise the properties of the data. Examples of descriptive data mining are clustering and association mining. In *predictive* data mining the goal is to learn a model that can predict values of unseen data based on the data provided. Examples of predictive data mining are classification and regression analysis. In this dissertation we focus on the descriptive data mining task. Although descriptive data mining algorithms can form the basis for a predictive algorithm, we will not consider this in this dissertation.

Another distinction can be made based on the data that is the input of the data mining task. We already mentioned itemset mining, but one can also consider for example sequence mining, tree mining, graph mining, and many others. In this dissertation we focus on *relational data mining*, where the considered input is a relational database. To be more precise, we focus on *descriptive relational data mining*, since the general relational data mining field also includes predictive data mining such as amongst others relational classification and decision tree learning [Džeroski, 2005].

The goal of this dissertation is to create efficient data mining algorithms that can find interesting (descriptive) *patterns* when the database in the knowledge discovery process is a *relational database.*

1.2 Relational Databases

A *database management system (DBMS)* is a software system that enables the creation, maintenance, and use of large amounts of data [Abiteboul et al., 1995]. *Relational* Database Management Systems (RDBMS) are based on the *relational model* [Codd, 1970], which has been the dominant paradigm for industrial database applications during the last decades, and it is at the core of all major commercial database systems, making RDBMSs one of the most commonly available kinds of data repositories.

A *relational database* is a collection of tables called *relations*, each of which is assigned a unique name. Each table consists of a set of *attributes* and usually stores a large set of *tuples.* Each tuple in a relational table represents an object identified by a unique *key* and described by a set of attribute values. Often one uses a semantic model to represent relational databases, allowing one to describe and design the database without having to pay attention to the physical database. Such a model is often referred to as a *database scheme.* One of the most common models is the *Entity-Relationship* (ER) model, which we will use in this dissertation. An Entity-Relationship model represents the database as entity sets and their relationships. An *entity set* is a collection of ‘similar’ entities. Each entity set has a set of proper-

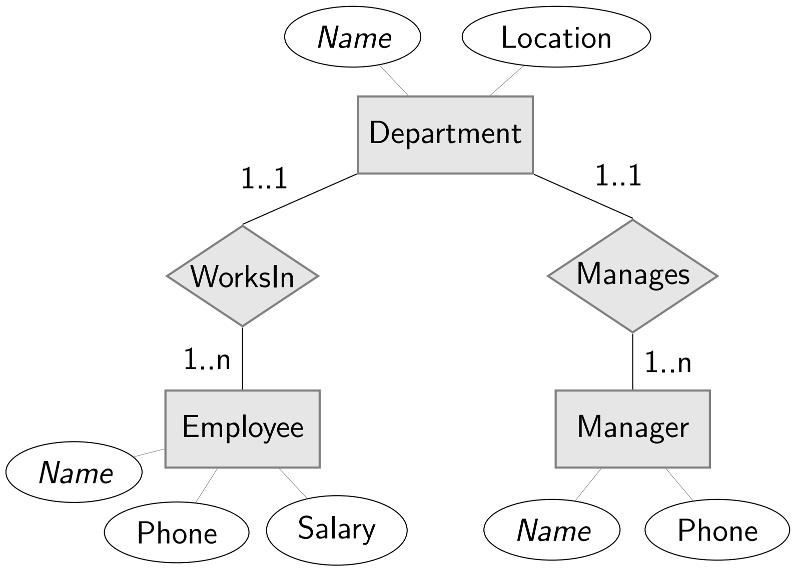


Figure 1.2: Example Entity-Relationship diagram

ties, called *attributes*. A subset of these attributes is called the *key*, and is unique for each entity in the entity set. Figure 1.2 shows an example Entity-Relationship diagram, the graphical notation used to represent Entity-Relationship models. In this diagram Employee is an entity set with attributes *Name*, Phone and Salary of which *Name* is the key. A *relationship* among entity sets is an ordered list of entity sets. The most common case is a list of two, *i.e.*, a binary relationship. For example, in Figure 1.2 we have the relationship Manages which connects the entity sets Department and Manager. Relationships also have an associated functionality, detailing how many entities from one entity set can be associated with how many entities from another entity set. In the example we have that a Department entity can have 1 to n (*one-to-many*) Manager entities associated, and that a Manager entity can only be associated with one Department entity (*one-to-one*). Note that these functionalities reflect the restrictions in the real world, and if a manager can manage more than one department, the model should also reflect this. It is clear, that an entity relationship model can easily be mapped onto a relational database, where both entity sets and relationships are represented as relations. An example relational database instance that follows the model is given in Figure 1.3.

To access or ‘query’ its database a DBMS provides a query language. In the case of an RDBMS this typically is SQL (Structured Query Language). SQL was one of the first languages developed for the relational model [Codd, 1970] and became the most widely used language for relational databases. In fact SQL is such a core aspect of many RDBMSs, that many of them have SQL in their name, e.g. MySQL, PostreSQL and SQLite. As a typical SQL query, consider the

Employee

Name	Phone	Salary
Eric House	2358	1400
Gregory Forman	4589	2000
Steven Moffat	4589	1800

Department

Name	Location
R&D	G
IT	A

Manager

Name	Phone
Jeff Haystack	1201
Susan Miller	1203

WorksIn

Employee	Department
Eric House	IT
Gregory Forman	R&D
Steven Moffat	R&D

Manages

Manager	Department
Jeff Haystack	IT
Susan Miller	R&D

Figure 1.3: Example relational database instance for the ER model of Figure 1.2

following query:

```
SELECT Employee.Name
FROM Employee, WorksIn, Department
WHERE Employee.Name = WorksIn.Employee
       AND Department.Name = WorksIn.Department
       AND Department.Name = "R&D"
       AND Employee.Salary > 1900
```

which asks for the names of the employees working in the department `R&D` that have a salary higher than `1900`. In the case of our example instance, this query will return the result:

Name
Gregory Forman

Since the goal of this dissertation is to create data mining algorithms that work on relational databases, our algorithms make use of the SQL language. This has the additional benefit that our algorithms are applicable to any RDBMS that supports SQL, and as already said, most of them do.

1.3 Overview

The goal of this dissertation is to create efficient data mining algorithms that can find interesting *patterns* in *relational databases*. To achieve this, we organised this dissertation as follows:

Chapter 2 introduces the problem of frequent pattern mining. Frequency is a basic constraint in pattern mining. Since data mining typically deals with huge volumes of data, in order for a pattern to be interesting it must hold for a large portion of this data. Hence it must occur frequently. We first take a look at frequent itemset mining, the simplest variant of frequent pattern mining. It provides an introduction to the basic properties and techniques that are also needed when mining more complex pattern kinds. We present the most import algorithms as well as how to avoid finding redundant itemsets. We then introduce a theoretic basis for the general problem of frequent pattern mining, and conclude by reviewing the area of relational pattern mining and situating our approaches in this context.

Chapter 3 introduces conjunctive queries as a pattern kind to be mined in relational databases. We define association rules over simple conjunctive queries as a simple, yet expressive pattern type and develop an efficient level-wise algorithm, Conqueror, to find interesting patterns. We then study some well-known dependencies that can be expressed by means of rules of simple conjunctive queries. Based on this knowledge we develop the Conqueror$^+$ algorithm, that discovers and makes use of these dependencies to efficiently compute non-redundant simple conjunctive queries.

Chapter 4 generalises itemset mining to relational databases. We define relational itemsets in such a way that the results are interesting and easy to interpret, contrasting it with existing approaches. We develop an efficient depth-first propagation-based algorithm to generate relational itemsets without redundancy. Furthermore, we introduce a new interestingness measure unique to the relational case, and show its benefits in mining relational databases.

Chapter 5 concludes this dissertation with a summary of the main contributions and some pointers towards further research.

Chapter 2

Frequent Pattern Mining

Frequent patterns are patterns (such as itemsets, subsequences or in general substructures) that appear frequently in a data set. Frequency is one of the basic interestingness measures used in pattern mining. Since data mining typically deals with huge volumes of data, only patterns that hold for a substantial amount of the data are considered potentially interesting. In this chapter we will first give an introduction to the area of frequent *itemset* mining and discus the two seminal algorithms. This way we introduce the concepts of frequent pattern mining in their historic setting before extending them to our intended setting of pattern mining in relational databases. Towards this goal, we conclude this chapter by first taking a closer look at the general frequent pattern mining problem and then at the specific approaches for mining frequent patterns in relational databases.

2.1 Frequent Itemset Mining

Frequent itemset mining was introduced in the context of the analysis of purchase-records of a supermarket [Agrawal et al., 1993], also referred to as market basket analysis. In this context we have a so-called transaction database, containing transactions of customers detailing the set of products they bought. A small example of such a database is given in Figure 2.1. The goal is to discover sets of products or *items* that are frequently bought together *i.e.*, the *frequent itemsets.* For example, in the database of Figure 2.1 we can see that 'bread' and 'butter' are frequently bought together. The supermarket management could use such results

TID	itemset
t_1	{butter}
t_2	{bread,butter}
t_3	{bread,chocolate}
t_4	{bread,butter,cheese}
t_5	{bread,cheese}
t_6	{bread,butter,chocolate}
t_7	{cheese}
t_8	{butter,cheese}
t_9	{bread,butter,cheese,ham}
t_{10}	{bread,butter,ham}

Figure 2.1: Example transaction database

to plan marketing or advertising strategies or special offers and sales. They could also be employed in designing different store layouts, for example by placing products that are frequently sold together near to each-other to encourage even more joint purchases. Alternatively, if there is a very strong correlation, one could also choose to put the products at the opposite ends of the store in order to entice customers to pick up other products along the route. Since its original introduction, itemset mining has been applied in many other applications domains and used for other purposes besides market basket analysis. Much of the terminology, however, still reflects these historic roots and we will therefore also use this context in our introductory examples.

Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be the set of possible items. A set $I = \{i_1, \ldots, i_k\} \subseteq \mathcal{I}$ is called an *itemset*, and if it contains k items, a *k-itemset*. Let $\mathcal{D}$ be a set of database transactions T, where each $T = (tid, I_{tid})$ is a couple where tid is the unique identifier associated with each transaction and I_{tid} is an itemset. A transaction $T = (tid, I_{tid})$ is said to *support* or contain an itemset I if and only if $I \subseteq I_{tid}$. The set of occurrences or *cover* of an itemset I in $\mathcal{D}$ is the set of transactions in $\mathcal{D}$ that support I:

$$cover_{\mathcal{D}}(I) = \{tid \mid (tid, I_{tid}) \in \mathcal{D}, I \subseteq I_{tid}\} \tag{2.1}$$

The *support* of an itemset I in $\mathcal{D}$ is the number of occurrences of the itemset in the database $\mathcal{D}$:

$$support_{\mathcal{D}}(I) = |cover_{\mathcal{D}}(I)| \tag{2.2}$$

In the example database of Figure 2.1 we have that *support*({bread,butter}) $= |\{t_2, t_4, t_6, t_9, t_{10}\}| = 5$. Note that here and in following examples we will leave out

$\mathcal{D}$ if it is clear from the context.

The *relative support* of an itemset I in $\mathcal{D}$, also sometimes referred to as *frequency*, is the support of I in $\mathcal{D}$ divided by the total number of transactions in $\mathcal{D}$. It corresponds to the probability of I occurring in a transaction $T \in \mathcal{D}$:

$$\mathit{frequency}_{\mathcal{D}}(I) = \frac{\mathit{support}_{\mathcal{D}}(I)}{|D|} = P(I) \tag{2.3}$$

In the example database we have $\mathit{frequency}(\{\text{bread,butter}\}) = 5/10 = 0.5$.

An itemset is called *frequent* if its support is no less than a given absolute *minimal support threshold*: minsup_{abs}, where $0 \leq \mathit{minsup}_{abs} \leq |\mathcal{D}|$. When working with relative support, we make use of a *minimal relative support threshold*: minsup_{rel}, where $0 \leq \mathit{minsup}_{rel} \leq 1$. It is clear that $\mathit{minsup}_{abs} = \lceil \mathit{minsup}_{rel} \times |\mathcal{D}| \rceil$. Typically we will just use one or the other and state this at the beginning, omitting the subscripts.

Definition 2.1. *Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$ and a minimal support threshold minsup, the set of* ***frequent items*** *over $\mathcal{D}$ is defined as:*

$$\mathcal{F}(\mathcal{D}, \mathit{minsup}) = \{I \subseteq \mathcal{I} \mid \mathit{support}_{\mathcal{D}}(I) \geq \mathit{minsup}\}$$

When $\mathcal{D}$ and minsup are clear from the context we will simply write $\mathcal{F}$.

The problem of frequent itemset mining can then be stated as follows:

Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$ and a minimal support threshold minsup find the set of frequent items $\mathcal{F}(\mathcal{D}, \mathit{minsup})$.

Frequent itemsets themselves are typically used as the basis of complex patterns. In the case of market basket analysis we are interested in *associated* items. In order to express this, the notion of an *association rule* was introduced.

Definition 2.2. *An* ***association rule*** *is an expression of the form $A \Rightarrow C$ where A and C are itemsets and $A \cap C = \{\}$.*

Such a rule expresses the association between transactions containing A and transactions containing C. A is called the *body* or *antecedent* of the rule, and C is called the *head* or *consequent* of the rule.

The confidence or accuracy of an association rule is the conditional probability of having C contained in a transaction, given that A is contained in that transaction:

$$\mathit{confidence}_{\mathcal{D}}(A \Rightarrow C) = P(C|A) = \frac{\mathit{support}_{\mathcal{D}}(A \cup C)}{\mathit{support}_{\mathcal{D}}(A)} \tag{2.4}$$

A rule is called *confident* if its confidence exceeds a given minimal confidence threshold *minconf* , where $0 \leq minconf \leq 1$. The support of a rule is the support of the union of antecedent and consequent:

$$support_{\mathcal{D}}(A \Rightarrow C) = support_{\mathcal{D}}(A \cup C) \tag{2.5}$$

In our example database the confidence of the rule {bread} ⇒ {butter} is:

$$confidence(\{\text{bread}\} \Rightarrow \{\text{butter}\}) = \frac{support(\{\text{bread,butter}\})}{support(\{\text{bread}\})} = \frac{5}{7} \approx 0.71$$

and the relative support, as computed before, is 0.5. The reasoning behind association rule mining is that rules with both a high support and a high confidence are very likely to reflect an interesting association. The example rule has a confidence of 71% with a support of 50% and could potentially be found to be interesting, as it implies 71% of the customers who buy bread also buy butter, and that this pattern is supported by 50% of the data.

Definition 2.3. *Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, a minimal support threshold minsup and a minimal confidence threshold minconf, the set of* ***frequent and confident association rules*** *over $\mathcal{D}$ is defined as:*

$$\begin{aligned}&\mathcal{R}(\mathcal{D}, minsup, minconf) = \\ &\qquad \{A \Rightarrow C \mid A, C \subseteq \mathcal{I}, A \cap C = \{\}, \\ &\qquad A \cup C \in \mathcal{F}(\mathcal{D}, minsup), confidence_{\mathcal{D}}(A \Rightarrow C) \geq minconf\}\end{aligned}$$

When $\mathcal{D}$, minsup and minconf are clear from the context we will simply write $\mathcal{R}$.

In general the problem of association rule mining can be stated as follows:

Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$ and a minimal support threshold minsup and a minimal confidence threshold minconf find $\mathcal{R}$($\mathcal{D}$, minsup, minconf).

Note that the frequent itemset mining problem is a special case of the association rule mining problem since any itemset I is represented by the rule $I \Rightarrow \{\}$ with 100% confidence. Additionally, all rules $A \Rightarrow C$ will hold with at least $minsup_{rel}$ confidence. The minimal confidence threshold should therefore be higher than the the minimal relative support threshold.

2.1.1 Mining Frequent Itemsets

The first algorithm developed to mine confident association rules ([Agrawal et al., 1993]) was divided into two phases. In the first phase all frequent itemsets are generated. The second phase is made up of the generation of all frequent and confident association rules. Many of the subsequent association rule mining algorithms also comply with this two phased strategy.

For a large number of items, it becomes infeasible to generate all itemsets and determine their support in order to find the frequent ones. That is, for $|\mathcal{I}|$ items there are $2^{|\mathcal{I}|}$ possible itemsets. The naive approach of finding all items quickly becomes intractable. For example, in the very typical case of a thousand items, the number of possible itemsets is approximately 10^{301}, which is already larger than the well know googol number (10^{100}) that in its turn is larger than the number of atoms in the observable universe ($\approx 10^{79}$). Of course we do not need to consider all possible itemsets and can limit ourselves to the itemsets that occur *at least once* in the transaction databases. Unfortunately for databases containing large transactions the number is mostly still too large. When generating itemsets we would ideally only want to generate the frequent ones. Unfortunately, this ideal solution is impossible in general. We will therefore have to consider several *candidate itemsets* and determine if these are frequent or not. Every considered candidate entails memory usage and computation time in order to obtain the support from the database. The goal is therefore to reduce the amount of candidate itemsets as much as possible in order to obtain an efficient algorithm. One property exploited by most of the itemset mining algorithms is the anti-monotonicity of support with respect to the set inclusion relation:

Proposition 2.1. ***(Support Anti-Monotonicity)***
Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, and two itemsets $I_1, I_2 \subseteq \mathcal{I}$ then if $I_1 \subseteq I_2$ it follows that $support_{\mathcal{D}}(I_2) \leq support_{\mathcal{D}}(I_1)$.

Proof. This follows trivially from the fact that the $cover_{\mathcal{D}}(I_2) \subseteq cover_{\mathcal{D}}(I_1)$. □

In our example database we have

$$\begin{aligned} support(\{\text{butter}\}) &= |\{t_2, t_4, t_6, t_8, t_9, t_{10}\}| = 6 \\ &\leq support(\{\text{bread,butter}\}) = |\{t_2, t_4, t_6, t_9, t_{10}\}| = 5 \end{aligned}$$

A consequence of this property is that

$$\text{if } support(I) < minsup \text{ then for all } I' \supset I : support(I') \leq support(I) < minsup.$$

This means that when searching for frequent itemsets we do not have to consider as candidates all supersets of an itemset that is infrequent, since we can derive their infrequency using the above property. When performing an incremental search for itemsets we can thus *prune* part of the search space by applying this property. The anti-monotonicity property is sometimes also referred to as the *Apriori property*, named after the *Apriori* algorithm [Agrawal & Srikant, 1994], improved version of the original algorithm of [Agrawal et al., 1993], in which this anti-monotonicity property was first exploited. The same technique was also independently proposed by [Mannila et al., 1994]. A joint paper was published afterwards [Agrawal et al., 1996]. In the following sections we introduce Apriori together with other important itemset mining algorithms.

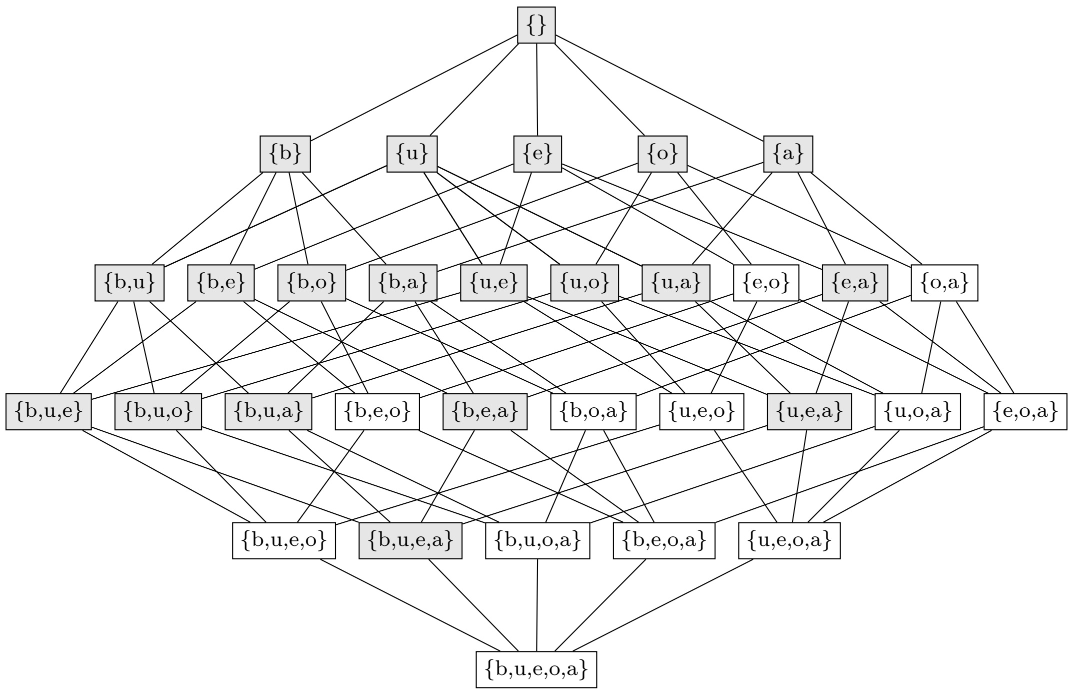


Figure 2.2: Search space for the example database from Figure 2.1 represented as a subset-lattice. Frequent itemsets for a minimal support of 1 are highlighted.

The search space of all itemsets can be represented as a *subset-lattice*, with the empty itemset as the top of the lattice, and the set containing all items as he bottom. A *Hasse diagram* is typically used to depict such a subset-lattice. For our example database the subset-lattice is represented in Figure 2.2 (note that we have abbreviated the product names **b**read, b**u**tter, ch**e**ese, ch**o**colate and h**a**m). In such a Hasse diagram each line represents a *direct* subset relation, *i.e.*, a line is drawn between I_1 and I_2 if and only if $I_1 \subset I_2$ and $|I_2| = |I_1| + 1$.

2.1.2 Mining Confident Rules

If one would need to consider all possible rules over a set of items $\mathcal{I}$, then one would need to consider $3^{|\mathcal{I}|}$ rules. Luckily we only need to consider those rules $A \Rightarrow C$

such that $A \cup C = I \subset \mathcal{F}$. For every such itemset I one can consider at most $2^{|I|}$ rules. It is clear that to efficiently traverse this search space, we need to iteratively generate and evaluate *candidate* association rules. The same techniques applicable in frequent itemset mining are also applicable in this case, and similar to support of itemsets we can also formulate a monotonicity property for confidence of rules.

Proposition 2.2. *Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, and let $A, B, C \subseteq \mathcal{I}$ be three itemsets such that $A \cap C = \{\}$. Then*

$$\mathit{confidence}_{\mathcal{D}}(A \setminus B \Rightarrow C \cup B) \leq \mathit{confidence}_{\mathcal{D}}(A \Rightarrow C)$$

Proof. Since $A \cup C \subseteq A \cup C \cup B$, and $A \setminus B \subseteq A$, we have

$$\frac{\mathit{support}_{\mathcal{D}}(A \cup C \cup B)}{\mathit{support}_{\mathcal{D}}(A \setminus B)} \leq \frac{\mathit{support}_{\mathcal{D}}(A \cup C)}{\mathit{support}_{\mathcal{D}}(A)}$$

□

This proposition states that confidence is monotone decreasing when moving an item from antecedent to consequent. This means that when considering association rules based on an itemset I if a certain $A \Rightarrow C$ such that $A \cup C = I$ is not confident, we do not need to consider any rules of the form $A \setminus B \Rightarrow C \cup B$. Similar to the anti-monotonicity of support this property is exploited in the algorithms shown in the next sections.

2.1.3 Algorithms

We will now detail two influential algorithms in frequent itemset mining; Apriori [Agrawal & Srikant, 1994] and Eclat [Zaki et al., 1997]. Since their original introduction, many optimisations and novel algorithmic adaptions have been proposed. We will, however, only cover these two original algorithms, since they introduce the basic algorithmic concepts of the frequent itemset mining setting. In later chapters we extend these concepts in order to apply them in the more general setting of mining relational databases.

Apriori

The Apriori algorithm is designed to maximise the use of Property 2.1. The itemset mining phase of the Apriori algorithm is shown in Algorithm 2.1. The algorithm uses a generate-and-test approach, and traverses the itemsets strictly increasing in size. In the initial pass over the database, the support values for all single items (singleton itemsets) are determined, and frequent single items identified. Then candidate itemsets are repeatedly generated and their frequency is determined.

Algorithm 2.1 Apriori

```
Input: database $\mathcal{D}$, $minsup$
Output: $\mathcal{F}$ set of frequent items
 1: $C_1 := \{\{i\} \mid i \in \mathcal{I}\}$
 2: $k := 1$
 3: while $C_k \neq \{\}$ do
 4:    //Compute the support values of all candidate itemsets
 5:    for all transactions $(tid, I) \in \mathcal{D}$ do
 6:       for all candidate itemsets $X \in C_k$ do
 7:          if $X \subseteq I$ then
 8:             $support(X)$++
 9:    //Extract all frequent itemsets
10:    $\mathcal{F}_k := \{X \mid support(X) \geq minsup\}$
11:    //Generate new candidate itemsets
12:    for all $X, Y \in \mathcal{F}_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ do
13:       $I = X \cup \{Y[k]\}$
14:       if $\forall J \subset I, |J| = k : J \in \mathcal{F}_k$ then
15:          $C_{k+1} := C_{k+1} \cup I$
16:    $k$++
```

We now give a more detailed description of this process. First of all we assume, without loss of generality, that the items in each itemset are ordered. We will assume a lexicographical order in our examples. In our pseudocode we will use the notation $I[i]$ to represent the i-th item of the itemset I. Given an itemset I we call the k-itemset $\{I[1], \ldots, I[k]\}$ the k-prefix of I. The Apriori algorithm uses a *breath-first* search (also called *level-wise* search) through the search space of all itemsets. It does this by iteratively generating sets of candidate $(k+1)$-itemsets C_{k+1}, based on the frequent k-itemsets $\mathcal{F}_k$. It only generates a candidate itemset if all of its subsets are known to be frequent (*i.e.*, we make use of Property 2.1). It generates a $(k+1)$-itemset by combining two k-itemsets that share a common $(k-1)$-prefix (line 12). In our example database of Figure 2.1, if we suppose a minimal support threshold of 1, the 3-itemset {bread,butter,cheese} would be generated using the 2-itemsets {bread,butter} and {bread,cheese}, that share the common 1-prefix {bread}. This way the generated itemset has at least two frequent subsets, and this also ensures all candidate itemsets are only generated once. The algorithm makes sure all subsets of a $k+1$ itemset are frequent by checking if all of its k-subsets are frequent (line 14). In our example this means that when considering the candidate itemset {bread,butter,cheese} the itemset {butter,cheese} is also checked. Note that we could avoid checking the generating subsets, but for clarity this is not represented in the pseudocode. In the case of minimal support 1 {butter,cheese}

Algorithm 2.2 Apriori Rule Generation

```
Input: database $\mathcal{D}$, $minsup$, $minconf$
Output: $\mathcal{R}$ set of frequent and confident rules
 1: Compute $\mathcal{F}(\mathcal{D}, minsup)$
 2: $\mathcal{R} := \{\}$
 3: for all $I \in \mathcal{F}$ do
 4:     $\mathcal{R} := \mathcal{R} \cup I \Rightarrow \{\}$
 5:     $C_1 := \{\{i\} \mid i \in I\}$
 6:     $k := 1$
 7:     while $C_k \neq \{\}$ do
 8:         //Extract all consequents of confident association rules
 9:         $H_k := \{X \in C_k \mid confidence(I \setminus X \Rightarrow X, \mathcal{D}) \geq minconf\}$
10:         //Generate new candidate consequents
11:         for all $X, Y \in H_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ do
12:             $I = X \cup \{Y[k]\}$
13:             if $\forall J \subset I, |I| = k : J \in H_k$ then
14:                 $C_{k+1} := C_{k+1} \cup I$
15:         $k$++
16:     //Cumulate all association rules
17:     $\mathcal{R} := \mathcal{R} \cup \{I \setminus X \Rightarrow X \mid X \in H_1 \cup \cdots \cup H_k\}$
```

is indeed found to be frequent. As a result, the itemset {bread,butter,cheese} is then added to C_3 in order for its support to be evaluated against the database (line 5). The candidate set C_1 is the base case, and here all singleton itemsets are added to the set of candidate items (line 1). For the set C_2 we assume a common empty prefix, and thus combine every pair of singleton itemsets to form the 2-itemsets.

Generating all frequent and confident confident rules can be done in much the same way as the generation of frequent itemsets and is presented in Algorithm 2.2. First of all, all frequent itemsets are generated using a frequent itemset mining algorithm (line 1). Then we divide every frequent itemset I into a consequent C and an antecedent $A = I \setminus C$. We first start with the empty head $\{\}$ thus generating rules of the type $I \Rightarrow \{\}$ (line 4). As stated we know these rules have 100% confidence. Then the algorithm iteratively generates candidate consequents of size $k+1$ based on the size k consequent of confident association rules. Similar to the frequent itemset mining case we generate $k+1$ size consequents by combining k size consequents of confident association rules that have a common $k-1$ prefix (line 11). Similarly we also check if all k-size subsets of the considered candidate are consequents of confident association rules (line 13), essentially using Property 2.2 to prune rules that are not confident. The candidate consequents are used to

TID	itemset
bread	$\{t_2, t_3, t_4, t_5, t_6, t_9, t_{10}\}$
butter	$\{t_1, t_4, t_6, t_8, t_9, t_{10}\}$
cheese	$\{t_4, t_5, t_7, t_8, t_9\}$
chocolate	$\{t_3, t_6\}$
ham	$\{t_9, t_{10}\}$

Figure 2.3: Example transaction database in vertical layout

compute the confidence of candidate rules and consequents resulting in confident rules stored in H_k. The computation of confidence itself can easily be achieved since we have already retrieved the collection of frequent itemsets on line 1, and thus we can easily obtain the support of I and X needed to compute $\mathit{confidence}(I \backslash X \Rightarrow X)$. In comparison to only mining frequent itemsets, the time required for finding all association rules given the frequent itemsets is relatively small. Finally all rules for itemset I are constructed using all H_k (line 17).

Eclat

As stated, the Apriori algorithm follows a breath-first approach. Alternatively one can also opt for a depth-first approach to frequent itemset mining. While the breadth-first approach tries to minimize the number of passes over the original database, a depth-first approach tries to optimise the speed of search. The first algorithm proposed to generate all frequent itemsets in a depth-first manner is the Eclat algorithm [Zaki et al., 1997]. This algorithm makes use of the so called *vertical database layout*. So far we have seen a transaction database as being stored by transaction identifiers associated with a set of items. This is referred to as the *horizontal database layout*. We can, however, also consider an alternative vertical database layout, where we store the single items together with the set of transactions in which they occur. Our example database in vertical database layout is given in Figure 2.3. The Eclat algorithm uses an intersection base approach to compute the support of an itemset, essentially intersecting the cover sets of two itemsets with a common prefix to obtain the cover set of the superset. For example we can compute the cover of {bread,butter} as follows:

$$\begin{aligned}
\mathit{cover}(\{\text{bread,butter}\}) &= \mathit{cover}(\{\text{bread}\}) \cap \mathit{cover}(\{\text{butter}\}) \\
&= \{t_2, t_3, t_4, t_5, t_6, t_9, t_{10}\} \cap \{t_1, t_4, t_6, t_8, t_9, t_{10}\} \\
&= \{t_4, t_6, t_9, t_{10}\}
\end{aligned}$$

Algorithm 2.3 Eclat

```
Input: database $\mathcal{D}$, $I \subset \mathcal{I}$, $minsup$
Output: $\mathcal{F}[I]$ set of frequent items having prefix $I$
 1: $\mathcal{F}[I] := \{\}$
 2: for all $i \in \mathcal{I}$ occurring in $\mathcal{D}$ do
 3:     $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
 4:     //Create $\mathcal{D}^i$
 5:     $\mathcal{D}^i := \{\}$
 6:     for all $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that $j > i$ do
 7:         $C := cover(\{i\}) \cap cover(\{j\})$
 8:         if $|C| \geq minsup$ then
 9:             $\mathcal{D}^i := \mathcal{D}^i \cup \{(j, C)\}$
10:     //Depth-first recursion
11:     Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, minsup)$
12:     $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
```

Although this method could also be used in a breath-first approach, it would require a large amount of memory, since we need to store the covers of each level. Therefore in this case a depth-first approach is more appropriate. We first note that we denote the set of all frequent k-itemsets with the same $(k-1)$-prefix represented by the itemset I as $\mathcal{F}[I]$. Eclat will recursively compute $\mathcal{F}[I]$ based on $F[I \cup \{i\}]$ for all $i \in \mathcal{I}$. Starting Eclat with $\mathcal{F}[\{\}]$ will then yield all frequent itemsets since $\mathcal{F}[\{\}] = \mathcal{F}$.

Eclat, given as Algorithm 2.3, considers the candidates $I \cup \{i, j\}$ for $i, j \in \mathcal{I}$ and $j > i$, thus ensuring no duplicates are generated (line 2 and 6). It computes the support by considering the intersection $cover(\{i\}) \cap cover(\{j\})$ (line 7) in database $\mathcal{D}$. For singleton itemsets (empty prefix) the considered database $\mathcal{D}$ is the original $\mathcal{D}$ in vertical database layout, and above an example of this case was given. When considering itemsets with a prefix I, however, the database for which the covers are computed is $\mathcal{D}^i$ where i is the last item of the prefix I. The database $\mathcal{D}^i$ is a *conditional* database, consisting of only those transactions that cover the itemset I. This means that for an item j the $cover(\{j\})$ in $\mathcal{D}^i$ is equal to the $cover(I \cup \{j\})$ in the original database $\mathcal{D}$. By recursively dividing the database in such a way (line 11), all frequent itemsets can eventually be discovered (line 12). Note that here the Apriori property is implicitly used since we only consider $\mathcal{D}^i$ of frequent itemsets I.

There are some drawbacks to this approach. First of all it does not fully exploit the anti-monotonicity property, and hence, the number of candidates generated is higher than the number of candidates generated in Apriori. For example, considering a minimal support of 3 and our example database, the candidate {bread,chocolate} will be considered by Eclat, although {chocolate} is not frequent. A second drawback is the fact that for *dense* databases, where many items occur in many transactions, the total size of the covers can become too large, even in a depth-first approach. [Zaki & Gouda, 2003] created the dEclat algorithm, an update of Eclat, to take this problem into account by using *diffsets*. Essentially instead of storing the full cover of a k-itemset I, only the difference between the cover of I and the cover of the $(k-1)$-prefix of I is stored. Many other optimisation have been proposed in later works, but we will not go into any more detail here. Another influential depth-first algorithm, which we will also not discuss in detail, is the FP-growth algorithm [Han et al., 2000, Han et al., 2004]. It is conceptually very similar to Eclat in the sense that it also uses conditional databases. However, instead of storing covers of itemsets, it stores the actual transactions of the database in a *prefix trie* structure, called the *FP-Tree*.

2.1.4 Closed Itemsets and Non-Redundant Rules

One problem that remains for all frequent pattern mining algorithms is the huge number of potentially interesting patterns they generate. In dense datasets one often has to use a low minimal support threshold in order to find interesting patterns, but this then leads to an explosion of the total output of frequent patterns. Some of these patterns can be considered redundant if they could have been derived from a smaller subset of the patterns. To tackle this problems, several subsets of the set of all frequent items have been defined that can act as a condensed or concise representation of the whole set. One of these subsets is the set of *closed itemsets* [Pasquier et al., 1999]. An itemset is called a closed itemset if all its supersets have a strictly lower support:

Definition 2.4. *Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$ and a minimal support threshold minsup, the set of* ***closed frequent items*** *over $\mathcal{D}$ is defined as:*

$$\mathcal{C}(\mathcal{D}, minsup) = \{I \in \mathcal{F}(\mathcal{D}, minsup) \mid \forall I' \supset I : support_{\mathcal{D}}(I') < support_{\mathcal{D}}(I)\}$$

Using only the closed frequent itemsets, we can determine the exact support value for all frequent itemsets. The support of a non-closed itemset is equal to the support of its smallest closed superset. For example the itemset {chocolate} is not closed since the support of its superset {bread,chocolate} is not smaller, it is in fact the same, namely 2. The {bread,chocolate} is a closed itemset since the support of its superset {bread,butter,chocolate} is smaller, namely 1. This smallest closed superset is defined as the *closure* of an itemset as follows:

$$closure_{\mathcal{D}}(I) = \bigcap_{tid \in cover_{\mathcal{D}}(I)} I_{tid}$$

Furthermore, we note that the support of an itemset is equal to that of its closure.

Proposition 2.3. *Given a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, for all $I \in \mathcal{I}$ it holds that*

$$support_{\mathcal{D}}(I) = support_{\mathcal{D}}(closure_{\mathcal{D}}(I))$$

This proposition essentially states that all frequent itemsets are uniquely determined by the set of frequent closed itemsets. Variants of the basic itemset mining algorithms to efficiently mine the closed itemsets include the *AClose* algorithm [Pasquier et al., 1999] based on Apriori, the CHARM algorithm [Zaki & Hsiao, 2002] based on Eclat, and the *Closet* algorithm [Pei et al., 2000] based on FP-Growth.

Next to the reduction of the number of itemsets, only considering closed itemsets also has benefits when considering association rules. [Zaki, 2000, Zaki, 2004] introduces a technique to mine non-redundant association rules based on closed itemsets. The set of association rules produced in this way is *generating*, in the sense that all possible association rules can be derived using operations like transitivity and augmentation. First let us define when a rule is redundant. Given a rule $R_1 = A_1 \Rightarrow C_1$, we say that the rule R_1 is more general than a rule $R_2 = A_2 \Rightarrow C_2$, denoted $R_1 \preceq R_2$ if we can generate R_2 by adding additional items to either the antecedent A_1 or consequent C_1 of R_1, *i.e.*, $A_1 \subseteq A_2$ and $C_1 \subseteq C_2$. A rule R is *redundant* if there exists a rule R' such that $R' \preceq R$ and $confidence_{\mathcal{D}}(R) = confidence_{\mathcal{D}}(R')$. In other words the non-redundant rules are those that are most general. Furthermore, it has been shown that association rules are transitive: if $I_1 \subseteq I_2 \subseteq I_3$ then if we know $I_1 \Rightarrow I_2$ has $x\%$ confidence and $I_2 \Rightarrow I_3$ has $y\%$ confidence we can deduce that the rule $I_1 \Rightarrow I_3$ has $xy\%$ confidence. Thus rules like $I_1 \Rightarrow I_3$ can also be considered redundant since these can be deduced.

Since the support of an itemset is equal to that of its closure, a rule $A \Rightarrow C$ is a equivalent variant of the rule $closure_{\mathcal{D}}(A) \Rightarrow closure_{\mathcal{D}}(C)$. So in order to generate all rules, it is sufficient to only consider all rules among closed itemsets. However, in a set of all rules that have the same closure, only the most general rules are non-redundant. The generation of these non redundant rules is based on the concept of *minimal generators* [Bastide et al., 2000].

Definition 2.5. *Let I be a closed itemset. We say that an itemset I' is a* ***generator*** *of I if and only if $I' \subseteq I$ and support(I') = support(I). I' is called a* ***proper generator*** *if $I' \subset I$.*

We note that a proper generator cannot be closed, since by definition a closed itemset cannot have a superset with the same support.

Definition 2.6. *Let $\mathcal{G}(I)$ denote the set of generators of I, then we say $I' \in \mathcal{G}(I)$ is a* ***minimal generator*** *if it has no subset in $\mathcal{G}(I)$. We use $\mathcal{G}^{\min}(I)$ to denote the set of all minimal generator of I.*

By definition $\mathcal{G}^{\min}(I) \neq \{\}$, since if there is no proper generator, an itemset I is its own minimal generator. For a closed itemset I it holds that its minimal generators are the minimal itemsets that are subsets of I but not subsets of any of I's immediate closed subsets, and they can be found using an Apriori style level-wise algorithm [Zaki, 2004]. For example, given the example database of Figure 2.1, the itemset {bread,chocolate} has one immediate closed subset: {bread}, thus the subset of {bread,chocolate} that is not a subset of {bread}, it the set {chocolate}, thus $\mathcal{G}^{\min}(\{\text{bread,chocolate}\}) = \{\{\text{chocolate}\}\}$.

Algorithm 2.4 Generate Non-Redundant Rules

```
Input: Database $\mathcal{D}$, $I_1, I_2 \in \mathcal{C}(\mathcal{D}, minsup)$, with $I_1 \subseteq I_2$
Output: $\mathcal{R}$ Set of non-redundant rules $A \Rightarrow C$ equivalent with $\mathcal{I}_1 \Rightarrow I_2$
 1: $G_1 := \mathcal{G}^{\min}(I_1)$
 2: $G_2 := \mathcal{G}^{\min}(I_2)$
 3: $\mathcal{R} := \{\}$
 4: //Rules with 100% confidence
 5: for all $I' \in G_2$ do
 6:   for all $I'' \in G_1$ do
 7:     $A := I'$
 8:     $C := I'' \setminus I'$
 9:     if $closure_{\mathcal{D}}(C) = I_1$ and $closure_{\mathcal{D}}(A \cup C) = I_2$ then
10:       $support_D(A \Rightarrow C) := support_D(I_2)$
11:       $confidence_D(A \Rightarrow C) := 1.0$
12:       $\mathcal{R} := \mathcal{R} \cup (A \Rightarrow C)$
13: //Rules with $< 100\%$ confidence
14: for all $I' \in G_1$ do
15:   for all $I'' \in G_2$ do
16:     $A := I'$
17:     $C := I'' \setminus I'$
18:     if $closure_{\mathcal{D}}(A \cup C) = I_2$ then
19:       $support_D(A \Rightarrow C) := support_D(I_2)$
20:       $confidence_D(A \Rightarrow C) := \frac{support_D(I_2)}{support_D(I_1)}$
21:       $\mathcal{R} := \mathcal{R} \cup (A \Rightarrow C)$
22: //Find all general rules
23: $\mathcal{R}^G := \{R_i \mid \nexists R_j \in \mathcal{R} : support_{\mathcal{D}}(R_j) = support_{\mathcal{D}}(R_i), confidence\mathcal{D}(R_j) = confidence\mathcal{D}(R_i), R_j \prec R_i\}$
```

The algorithm for rule generation based on the notion of minimal generators is shown as Algorithm 2.4 [Zaki, 2004]. It first computes the minimal generators for both closed itemsets (line 1 and 2). Then it checks if there are any 100% rules that hold by considering the minimal generator variants of $I_2 \Rightarrow I_1$, hereby making sure antecedent and consequent are disjunct (line 8) and that the rule is equivalent to $I_2 \Rightarrow I_1$ (line 9). Then it checks if there are any $<$ 100% rules that hold by considering the minimal generator variants of $I_1 \Rightarrow I_2$, also again ensuring the disjunction and equivalence. Finally, from this final set of rules, any redundant non most-general rules are eliminated (line 23)

Example 2.1. *Considering the example database from Figure 2.1, let the closed itemsets {bread,chocolate} and {bread,butter,chocolate} be the input to the algorithm. First we compute the minimal generators resulting in* $G_1 = \{\{\mathit{chocolate}\}\}$ *and* $G_1 = \{\{\mathit{butter,chocolate}\}\}$*. For candidate* 100% *confidence rules we consider {butter,chocolate}* $\Rightarrow$ *{butter}. However,*

$$\mathit{closure}_{\mathcal{D}}(\{\mathit{butter}\}) = \{\mathit{butter}\} \neq I_1 = \{\mathit{bread,chocolate}\}\ .$$

Thus we have no 100% *rules. For the* $<$ 100% *confidence rules we consider the rule {chocolate}* $\Rightarrow$ *{butter}. Here it does hold that*

$$\mathit{closure}_{\mathcal{D}}(\{\mathit{chocolate,butter}\}) = \{\mathit{bread,butter,chocolate}\} = I_2.$$

The confidence of {chocolate} $\Rightarrow$ *{butter} is* 50%*, and the support is* 1*. Thus* $\mathcal{R}^G = \mathcal{R} = \{(\{\mathit{chocolate}\} \Rightarrow \{\mathit{butter}\})\}$*.*

2.2 Frequent Pattern Mining

Although databases consisting of transactions of items can be used for various purposes other than market basket analysis, there is a need for algorithms that can mine other types of data. Other types of data, require other types of patterns. [Mannila & Toivonen, 1997] showed that the basic elements of frequent itemset mining can be generalised to mine any type of patterns. In their framework they describe the task of finding all potentially interesting sentences (patterns) as follows: Assume a database $\mathcal{D}$, a language $\mathcal{L}$ that can express properties or define subgroups of the data in database $\mathcal{D}$, and a selection predicate q are given. The predicate q is used for evaluating whether a sentence $\varphi \in \mathcal{L}$ defines a potentially interesting subclass of $\mathcal{D}$. The task is then to find the theory of $\mathcal{D}$ with respect to $\mathcal{L}$ and q, *i.e.*, the set $\mathit{Th}(\mathcal{L}, \mathcal{D}, q) = \{\varphi \in \mathcal{L} \mid q(\mathcal{D}, \varphi) \text{ is true}\}$

Example 2.2. *For frequent itemset mining we have* $\mathcal{D}$ *consisting of sets of items (the transaction database), the language* $\mathcal{L}$ *expresses all subsets* I *of elements of*

$\mathcal{D}$ (the itemsets) and the selection predicate $q(\mathcal{D}, I)$ is true only if $support_D(I) \geq minsup$.

Note that in this general framework the selection predicate q is left unspecified. The predicate $q(\mathcal{D}, I)$ could for example mean that φ is true or almost true in $\mathcal{D}$, or φ in some way defines an interesting subgroup of $\mathcal{D}$ (e.g. the frequent itemsets). In *frequent* pattern mining, the case we are considering in this chapter, $\mathcal{L}$ will describe patterns in the data (typically subsets of the data), and $q(\mathcal{D}, I)$ will be a frequency constraint. Note, however, that some additional constraints can be added on top of that.

Next to this formalisation of the general pattern discovery task, [Mannila & Toivonen, 1997] also define a general level-wise algorithm for computing the collection $\mathcal{T}h(\mathcal{L}, \mathcal{D}, q)$. It is based on the existence of a specialization relation on the sentences in $\mathcal{L}$ that is monotone with respect to q. A *specialisation relation* is a partial order $\preceq$ on the sentences in $\mathcal{L}$.

Definition 2.7. *A binary relation $\preceq$ is a **partial order** over a set $\mathcal{L}$ if $\forall \varphi, \varphi', \varphi'' \in \mathcal{L}$ it holds that:*

1. *$\varphi \preceq \varphi$ (reflexivity)*
2. *if $\varphi \preceq \varphi'$ and $\varphi' \preceq \varphi$ then $\varphi = \varphi'$ (anti-symmetry)*
3. *$\varphi \preceq \varphi'$ and $\varphi' \preceq \varphi''$ then $\varphi \preceq \varphi''$ (transitivity)*

We say that φ is more general than φ' or that φ' is more specific than φ if $\varphi \preceq \varphi'$. Note that the subset relation on itemsets $\subseteq$ and the order defined on association rules in Section 2.1.4 are both examples of partial orders. A specialisation relation $\preceq$ is a *monotone* specialization relation with respect to q of the selection predicate q is monotone with respect to $\preceq$.

Definition 2.8. *A predicate q is **monotone** with respect to $\preceq$ if for all $\mathcal{D}$ and φ we have that:*

$$\textit{if } q(\mathcal{D}, \varphi) \textit{and } \varphi' \preceq \varphi \textit{ then } q(\mathcal{D}, \varphi')$$

As was shown in Section 2.1, the subset relation $\subseteq$ is monotone with respect the predicate $q(\mathcal{D}, I) = (support_D(I) \geq minsup)$, since an itemset can only be frequent if all of its subsets are frequent.

The general level-wise algorithm presented by [Mannila & Toivonen, 1997] is shown as Algorithm 2.5. It is clear that this algorithm is a generalisation of the Apriori algorithm from Section 2.1.3, as it shares most of the structure. The algorithm works iteratively, alternating between *candidate generation* and *candidate evaluation*, and is the prototype of a so called generate-and-test approach. In the generation phase of an iteration k a collection C_k of new candidates sentences is generated, making use of the information available from more general sentences (line 7). Then the selection predicate is evaluated on these candidate sentences, and satisfying candidates are kept in $\mathcal{F}_k$ (line 5) . To start off with, the algorithm constructs C_1 to contain all the most general sentences (line 1). The iteration stops when no more potentially interesting sentences can be found. The algorithm is designed to minimise the amount of database processing, *i.e.*, the number of evaluations of q, by exploiting the monotonicity property.

Algorithm 2.5 Level-wise Pattern Mining Algorithm

```
Input: Database $\mathcal{D}$, language $\mathcal{L}$, as specialisation relation $\preceq$ monotone with respect to a selection predicate $q$
Output: The set $\mathcal{T}h(\mathcal{L}, \mathcal{D}, q)$
1: $C_1 := \{\varphi \in \mathcal{L} \mid$ there is no $\varphi' \in \mathcal{L}$ such that $\varphi' \preceq \varphi\}$
2: $k := 1$
3: while $C_k \neq \{\}$ do
4:     //Evaluation: find which sentences of $C_k$ satisfy $q$
5:     $\mathcal{F}_k := \{\varphi \in C_k \mid q(\mathcal{D}, \varphi)\}$
6:     //Generation: compute $C_{k+1} \subset \mathcal{L}$ using $\bigcup_{l \leq k} \mathcal{F}_l$
7:     $C_{k+1} := \{\varphi \in \mathcal{L} \mid \forall \varphi' : \varphi' \preceq \varphi$ and $\bigcup_{l\ leqk} \mathcal{F}_l\} \setminus \bigcup_{l \leq k} C_l$
8:     $k{+}{+}$
```

As already stated, in the case of *frequent* pattern mining this general algorithm could be further specified since we will be assuming we are dealing with frequency as the selection predicate q. In general, we can conclude that if we are able to define a specialisation relationship on our pattern type such that it is monotone with respect to the frequency predicate, we are able to use the level-wise approach to mine frequent patterns of this type. This fact has resulted in many level-wise algorithms that mine patterns more complex than itemsets, e.g. sequences [Agrawal & Srikant, 1995], trees [Zaki, 2002] and graphs [Kuramochi & Karypis, 2001]. Of course, as with itemsets, other non-level-wise strategies have also been devised for complex patterns, such as the depth-first *gSpan* [Yan & Han, 2002] for graphs, and TreeMiner for trees [Zaki, 2002]. It must be noted that many of these approaches still work with *transaction-like* databases, in the sense that instead of itemsets these transactions now contain sequences, graphs or trees. For example, in the tree case [Zaki, 2002], every transaction in the database contains a tree, and the presented algorithm tries to find all frequent subtrees occurring within all such transactions.

When we consider mining for frequent *relational* patterns, the input database is clear: a relational database. However, an arbitrary relational database is very different in structure than a transactional database, even compared to a transaction-like database containing complex data structures. Therefore, in the case of a relational database, the elements of the general frequent pattern mining task like the pattern language $\mathcal{L}$, the specialisation relation $\preceq$ and even q are not straightforwardly defined. In the next section we will take a closer look at the possibilities.

2.3 Relational Pattern Mining

As discussed in Chapter 1, a relational database contains multiple entity sets and relationships connecting them. Because of this complex structure, one can define many different kinds of patterns on this data. In the literature several options have been studied.

Relational itemset mining is a generalization of itemset mining on transactional databases to itemset mining on relational databases. Before we discuss arbitrary relational databases consisting of multiple tables, let us first consider the case of a relational database consisting of only one table. Even in this case, one cannot simply apply standard itemset mining techniques. As stated in Chapter 1 a relational table has an associated set of attributes, and the tuples of this table have different values for all these attributes. Regular itemset mining can be seen as a specific case of mining in a single relational table, where the attributes considered are all possible items and each transaction corresponds to a tuple where we have the value '1' for the attribute if the item is present in the transaction and '0' if it is not. In other words, we have a *binary* dataset. Itemset mining is therefore also often consider to be mining of binary datasets. A general relational table, however, has attributes that can have more than two different values. One solutions that allows us to apply itemset mining techniques on a single relational table is to consider attribute-value pairs as single items [Srikant & Agrawal, 1996]. Using this approach one can actually transform a single relational table into a transactional database, by converting each tuple into a set of attribute-value items. In general, attributes can be quantitative or categorical. Only considering categorical attributes, this transformation is relatively straightforward, since only a limited number of values is possible. Quantitative attributes, however, are defined over infinite domains. One way of dealing with this is discretisation [Liu et al., 2002], which essentially allows us to convert quantitative into categorical attributes, but other approaches like distribution [Webb, 2000] or optimisation [Brin et al., 2003] based techniques are also being investigated. Since the focus of this dissertation is not on these issues, we assume we are always dealing with categorical attributes.

Despite these possibilities for mining in a single relational table, an arbitrary relational database is not made up of a single table. It contains several tables, representing entities and their relations. As such, an arbitrary relational database cannot be trivially converted to a transactional database without loosing information. The approach taken in *relational itemset* mining, is to join all the tables of the database into one big table. This brings us back to the one-table case, and thus we can apply adapted itemset mining techniques [Crestana-Jensen & Soparkar, 2000, Ng et al., 2002, Koopman & Siebes, 2008]. Nevertheless, there are some issues related to this technique regarding support counting. In Chapter 4 we provide solutions to these problems by introducing a new definition of *relational itemset* together with an efficient depth-first algorithm for mining them.

Early on *queries* have been proposed as a natural pattern type for relational databases [Dehaspe & Raedt, 1997, Dehaspe & Toivonen, 1999, Dehaspe & Toivonen, 2001]. Since queries are widely used in order to retrieve subsets of data from the databases, they can also be regarded as descriptions for these subsets of data. Hence, queries form a natural pattern language $\mathcal{L}$. Furthermore, association rules over queries are also easily defined by considering queries and more specific queries (containing more restrictions). To illustrate, consider the well known Internet Movie Database [IMDB, 2008] containing almost all possible information about movies, actors and everything related to that, and consider the following queries: first, we consider the query that asks for all actors that have starred in a movie of the genre 'drama'; then, we can also consider the more specific query that asks for all actors that have starred in a movie of the genre 'drama', but that also starred in a (possibly different) movie of the genre 'comedy'. Now suppose the answer to the first query consists of 1000 actors, and the answer to the second query consists of 900 actors. Then this association rule reveals the potentially interesting pattern that actors starring in 'drama' movies typically (with a probability of 90%) also star in a 'comedy' movie. Of course, this pattern could also have been found using itemset mining techniques. But one would have to first transform the database, and create a transaction for each actor containing the set of all genres of movies he or she appeared in. Similarly, a pattern like: 77% of the movies starring Ben Affleck, also star Matt Damon, could be found by considering the query asking for all movies starring Ben Affleck, and the query asking for all movies starring both Ben Affleck and Matt Damon. Again, this could also be found using frequent itemset mining methods, but this time, the database should have been differently preprocessed in order to find this pattern. Furthermore, it is even impossible to preprocess the database only once in such a way that the above two patterns would be found by frequent set mining as they are essentially counting a different type of transactions. Indeed, we are counting actors in the first example, and movies in the second example. Also truly relational patterns can be found which cannot be found using typical set mining techniques, such as, 80% of all movie directors that have ever been an actor in some movie, also star in at least one of the movies they directed themselves. This can be easily expressed by two simple queries of which one asks for all movie directors that have ever acted, and the second one asks for all movie directors that have ever acted in one of their own movies. In general, we are looking for association rules $Q_1 \Rightarrow Q_2$, such that Q_1 asks for a set of tuples satisfying a certain condition and Q_2 asks for those tuples satisfying a more specific condition.

Of course there does not exist only one type of query, many types of queries and many query languages exist. Furthermore, the choice of query language $\mathcal{L}$ will also determine the specialisation relation $\preceq$ as well as the selection predicate q. In Chapter 3 we discuss queries as a pattern language and introduce our own query based approach for mining relational databases.

Next to relational itemsets and queries, several other types of patterns have been considered in the field. Typically these pattern types pose certain restrictions on the relational databases on which they can be applied. For example attribute-trees, introduced by [De Knijf, 2007] are only applicable to relational databases with a tree-shaped scheme. The tree-like 'relational patterns' by [Tsechansky et al., 1999], are only applicable to databases in the hierarchical model (which, we must note, cannot be fully mapped to the relational model [Ullman, 1988], and multi-dimensional patterns, introduced by [Kamber et al., 1997], are only applicable to relational databases with a star-scheme.

2.4 Conclusion

In this chapter we have provided an introduction to the field of frequent pattern mining. We have provided an overview of frequent itemset mining, and its most influential algorithms: the breadth-first Apriori algorithm and the depth-first Eclat algorithm. These algorithms provide insight into the origins of the techniques that we introduce in the next chapters for mining patterns in relational databases. Furthermore, we also touched on the problem of redundancy in itemsets and rules. This problem resurfaces when considering patterns in relational databases, and we show similar techniques can be applied. Furthermore, we have considered the theoretical general level-wise framework for frequent pattern mining, which forms the basis of many more specialised pattern mining algorithms, including the frequent query mining algorithm we introduce in Chapter 3. To conclude, we gave a brief overview of the pattern types used in the relational pattern mining field, including relational itemsets which we cover in more detail in Chapter 4 and queries detailed in Chapter 3.

Chapter 3

Conjunctive Query Mining

Queries are a basic concept in any database paradigm. In general they are used both for retrieving as well as managing data in databases. Database queries can be written in a variety of languages, sometimes specific to the database system used. Probably the most widely known and adopted query language is SQL, the Structured Query Language, designed for use with relational database management systems (RDBMSs). It is supported in all of the major commercial and non-commercial RDBMSs. Apart from querying of data, it also includes data updates and database scheme creation and modification, making SQL a complex but powerful language. In order to define and study the theoretical basis of relational query languages in general, several abstract languages have been developed. In the relational algebra, algebraic notations are used to express queries by applying specialised operators to relations. In the relational calculus, queries are expressed by writing logical formulas that the tuples in the answer must satisfy. The relational calculus is a restriction of the query language datalog (essentially without recursion), which, in turn, is a subset of first-order (predicate) logic. Datalog is derived from the logic programming language prolog. Although datalog as a query language is popular in academic database research, it has never succeeded in becoming part of commercial database systems, even though it has some advantages (compared to SQL) such as recursive queries and concise semantics.

As stated in Chapter 2, we are interested in the discovery of frequent patterns in relational databases. Current algorithms for frequent pattern discovery mainly focus on transactional databases. Even though complex structures like trees or graphs are considered, the algorithms typically still work on sets of transactions. Let us consider an example from the tree case [Zaki, 2002]. Here every transaction is a single tree, and the algorithm mines all frequent subtrees occurring in all such transactions. In order to use these types of algorithms one is required to transform the orignal data in the relational database into transactions. There is, however, no single way to do this for every arbitrary database, and even if we choose one such transformation, a lot of information implicitly encoded in the relational model would be lost. In that respect we recall the example of the movie database introduced in Chapter 2, where we showed two different transformations would be needed to find patterns regarding movies versus patterns regarding actors.

Furthermore, typical frequent pattern mining algorithms use several specialised data structures and indexing techniques to efficiently find their specific kind of pattern, be it itemsets, trees, graphs or something else. Since a database query is typically used to retrieve a subset of the data contained in a relational database, such a query can also be used to describe this specific subset of the data, and thus can be considered as a pattern. In this chapter we explore how to use queries as a language to describe frequent patterns in arbitrary relational databases, and create an efficient algorithm to find them.

Since our goal is to create algorithms that can operate directly on existing RDBMSs, using SQL as a pattern language clearly is the best choice. For our formal query notation, however, we make use of the relational algebra as it, in our opinion, matches SQL, which we use in practice, most closely. Considering all SQL or relational algebra queries, however, would yield an immensely large pattern space. In a generate-and-test strategy (see Chapter 2), trying to generate all such queries without any further restriction would be infeasible. Although there are various options to create a smaller pattern space (e.g. allowing the user to fully specify exactly which kind of queries he is interested in), we choose to restrict ourselves to a well studied class of relational algebra queries: the select-project-join queries. These are the relational algebra queries that do not use the operations union or difference, corresponding to basic select-from-where queries in SQL of which the where-condition consists exclusively of conjunctions of equality conditions. Equality conditions are those conditions constructed from attribute names and constants only using the “=” comparison operator. As an example we can consider the following select-from-where SQL query:

```
SELECT stars.actor
FROM stars, genre
WHERE stars.movie = genre.movie
      AND genre.genre = "comedy"
```

The same query in relational algebra, the notation we use throughout this chapter, looks like:

$$\pi_{\text{stars.actor}}\sigma_{\text{stars.movie=genre.genre}\wedge\text{genre.genre=`comedy'}}(\text{stars} \times \text{genre})$$

In first-order logic the select-project-join queries are the queries that can be constructed only using atomic formulae, the conjunction ($\wedge$) and existential quantification ($\exists$). This results in the following formula:

$$(x_1, \ldots, x_n).\exists x_{n+1}, \ldots, x_m.A_1 \wedge \ldots \wedge A_p$$

where $x_1, \ldots, x_n$ are free variables, and $x_{n+1}, \ldots, x_m$ are the bound variables and $A_1 \wedge \ldots \wedge A_p$ are atomic formulae. Because of this notation, these queries are more commonly referred to as the *conjunctive queries*, and we will also use this name. The example query can also be written in first-order logic as follows:

$$(\text{actor})\ .\ \exists\text{movie}\ .\ \text{stars}(\text{movie}, \text{actor}) \wedge \text{genre}(\text{movie}, \text{`comedy'})$$

As mentioned, we can also write queries as datalog rules, our example query can be posed as the datalog query

$$q(\text{actor}) \text{ :- } \text{stars}(\text{movie}, \text{actor}), \text{genre}(\text{movie}, \text{`comedy'})$$

Although conjunctive queries look simple, a large part of the queries that are typically issued on relational databases can be written as conjunctive queries. Furthermore, conjunctive queries have a number of desirable theoretical properties that the general class of all relational algebra queries do not share.

One important problem, that we also show to be relevant in our data mining context, is the query containment problem, *i.e.*, deciding if two queries are *contained*, where containment is defined as follows [Ullman, 1989]:

Definition 3.1. *For two queries Q_1 and Q_2 over a relational scheme $\mathcal{D}$, we say that $Q_1 \subseteq Q_2$ (Q_1 is **contained** in Q_2) if for every possible instance $\mathcal{I}$ it holds that the result of Q_1 evaluated over the instance $\mathcal{I}$, denoted by $Q_1(\mathcal{I})$, is contained in result of Q_2 over the same instance (denoted $Q_2(\mathcal{I})$). Q_1 and Q_2 are called* equivalent*, denoted $Q_1 \equiv Q_2$, if and only if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.*

While deciding containment, and therefore also equivalence, is undecidable for relational algebra and SQL queries, it is decidable but NP-complete for conjunctive queries [Chandra & Merlin, 1977]. Algorithms to solve the problem can take time that is exponential in the size of the input, that is, the length of the conjunctive queries. For the important subclass of the acyclic conjunctive queries, deciding query containment is possible in polynomial time [Yannakakis, 1981, Chekuri & Rajaraman, 2000]. Acyclic conjunctive queries are those conjunctive queries that can be represented using an acyclic hypergraph. For our frequent pattern mining purpose, we introduce a new, even stricter subclass of conjunctive queries, called the *simple conjunctive queries*, of which we show query containment can be decided in linear time (Section 3.1). Despite the class being very strict, we show that many interesting and even well known kinds of patterns can be discovered. Amongst them are database dependencies (Section 3.4), such as functional and inclusion dependencies. Next to a basic algorithm that efficiently discovers simple conjunctive queries (Section 3.2 and 3.3), we create a more advanced algorithm that takes dependencies into account and does not generate any queries redundant with respect to these dependencies (Section 3.6 and 3.7). Since, as mentioned, we are also able to *detect* dependencies, we create this algorithm in such a way that it is capable of instantly using newly discovered dependencies. Several experiments clearly show the benefits of discovering and using. Our approach makes the discovery of simple conjunctive queries a feasible and attractive method towards the exploration of arbitrary relational databases. Part of this chapter is based on work published earlier in [Goethals et al., 2008].

3.1 Simple Conjunctive Queries

As stated before, the basic strategy used in frequent pattern mining is the generate-and-test approach. This requires a pattern class that can be generated efficiently. With this requirement in mind, we define the novel class of *simple conjunctive queries*, together with a partial order, which allows us to generate them efficiently, as well as to define association rules.

Assume we are given a relational database consisting of a relational scheme $\mathcal{D} = \{R_1, \ldots, R_n\}$ over a fixed attribute set U, such that for $i = 1, \ldots, n$, R_i is a relation name associated with a subset of U, called the *scheme* of R_i and denoted by $sch(R_i)$. Without loss of generality, we assume that, for all distinct i and j in $\{1, \ldots, n\}$, $sch(R_i) \cap sch(R_j) = \emptyset$. In order to make this assumption explicit, for all i in $\{1, \ldots, n\}$, every A in $sch(R_i)$ is referred to as $R_i.A$.

Definition 3.2. *A* ***simple conjunctive query*** *Q over $\mathcal{D}$ is a relational algebra expression of the form*

$$\pi_X \sigma_F (R_1 \times \cdots \times R_n),$$

with X a set of attributes from $R_1, \ldots, R_n$ denoted $\pi(Q)$ and $F = \bowtie(Q) \wedge \sigma(Q)$, where $\bowtie(Q)$ is a conjunction of selection conditions of the form $R_i.A = R_j.A'$, and $\sigma(Q)$ of conditions of the form $R_k.A = c$, where i, j and k are in $\{1, \ldots, n\}$, $R_i.A, R_jA', R_k.A \in U$, and where c is a constant from the domain of $R_k.A$. The conditions $\sigma(Q)$ define a tuple which we denoted by Q^σ.

The only simplification, although drastic, over general conjunctive queries, is that every relation from $\mathcal{D}$ occurs exactly once in a simple conjunctive query. Furthermore, for our query evaluation, we always assume *duplicates are eliminated*. Knowing this, we could loosen the definition to allow every relation at most once, but this would unnecessarily complicate matters. After all, under the additional assumption that all relations are non-empty, both definitions are equivalent, as illustrated in the following example.

Example 3.1. *Let us consider a database schema $\mathcal{D}$ consisting of two relation names R_1 and R_2 with the following schemas: $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$. Since $sch(R_1) \cap sch(R_2)$ is clearly empty, in this example and in the forthcoming examples dealing with $\mathcal{D}$, we do not prefix attributes with relation names. For example, $R_1.A$ is denoted by A. Assuming all the relations to be non-empty, the following queries result in the same set of unique tuples under duplicate elimination:*

$$Q_1 : \pi_{A,B}(R_1)$$
$$Q_2 : \pi_{A,B}(R_1 \times R_2)$$

Therefore, we only consider the second type as simple conjunctive queries.

3.1.1 Query Comparison

The motivation for defining a query comparison relation is twofold. Firstly, an order allows for efficient generation of our pattern class, the simple conjunctive queries, allowing us to adopt a levelwise strategy when creating an algorithm (see Chapter 2). Secondly, although some interesting facts can be discovered using only simple conjunctive queries, the results are much more interesting when queries are compared to each other. For that reason, the comparison of two queries forms the basis of our definition of association rules. We first expand on the latter, starting with the following motivating example.

Example 3.2. *Assuming the database schema $\mathcal{D}$ from Example 3.1, consider the following two queries:*

$$Q_1 : \pi_{A,B}(R_1 \times R_2)$$
$$Q_2 : \pi_{A,B}\sigma_{A=B}(R_1 \times R_2)$$

Now suppose Q_1 returns 100 unique tuples, and Q_2 returns 90 unique tuples. Comparing these queries is potentially interesting as combined they represent the pattern that for 90% of the tuples in Q_1, the value of attribute A equals the value of attribute B.

Generalising we can state that we are interested in those pairs of simple conjunctive queries Q_1 and Q_2 stating that a tuple in the result of Q_1 is also in the result of Q_2 with a given probability p. Finding such pairs of queries comes down to finding pairs of *contained* simple conjunctive queries and testing if they meet the probability requirement. Note that following the standard definition of containment, we do not test containment for queries of which the projections are different. Queries with different projections describe conditions on different sets of tuples and hence cannot result in rules relating these conditions. Given the setting of Example 3.1, we therefore would not test if $\pi_A(R_1 \times R_2) \subseteq \pi_C(R_1 \times R_2)$. However, these kind of patterns can always be expressed by patterns of queries having the same projection, in this case by the test $\pi_A(R_1 \times R_2) \equiv \pi_A\sigma_{A=C}(R_1 \times R_2)$.

Nevertheless, there is one case in which comparing simple conjunctive queries with a different projection does produce interesting results, namely when one projection is a subset of the other. As this is not included in the classical definition of containment, we consider it as a second containment relationship between conjunctive queries. The following example illustrates this.

Example 3.3. *Consider the following two queries.*

$$\begin{aligned} Q_1 &: \pi_{A,B}(R_1 \times R_2) \\ Q_2 &: \pi_A(R_1 \times R_2) \end{aligned}$$

Now suppose Q_1 returns 100 unique tuples, and Q_2 returns 90 unique tuples (due to duplicate elimination). Then, again this pair of queries is potentially interesting as they represent the pattern that the number of unique values for A equals 90% of the number of unique tuples in Q_1. This implies we can choose 90% of the unique tuples in Q_1 such that these all have a unique A value. Note, however, we do not know if the remaining 10% have the same value for A, or whether they all have different values also occurring in the other 90%.

This second type of containment, we call *vertical* containment. Indeed, when projecting on a subset of the attributes in the projection of a query, the result is *vertically* contained in the result of the original query. As we show later, interesting, and even well known associations can be found using this *vertical* containment. When both types of containment are combined, we arrive at our formal containment definition which we use to discover simple conjunctive queries [Goethals & Van den Bussche, 2002].

Definition 3.3. *Let Q_1 and Q_2 be two simple conjunctive queries, we say $Q_1 \subseteq^\Delta Q_2$ (Q_1 is* ***diagonally contained*** *in Q_2) if $\pi(Q_1) \subseteq \pi(Q_2)$ and $Q_1 \subseteq \pi_{\pi(Q_1)}Q_2$.*

Using this we now formally define association rules over simple conjunctive queries:

Definition 3.4. *An* ***association rule*** *is of the form $Q_1 \Rightarrow Q_2$, such that Q_1 and Q_2 are both simple conjunctive queries and $Q_2 \subseteq^\Delta Q_1$.*

As in general frequent pattern mining, we are not interested in all possible association rules, but only in those that satisfy certain requirements. The first requirement states that an association rule should be *supported* by a minimum number of tuples in the database.

Definition 3.5. *The* ***support*** *of a simple conjunctive query Q in an instance $\mathcal{I}$ of $\mathcal{D}$, denoted support$_{\mathcal{I}}(Q)$ or simply support(Q), is the number of unique tuples in the answer of Q on $\mathcal{I}$. A simple conjunctive query is said to be* frequent *in $\mathcal{I}$ if its support exceeds a given minimal support threshold. The support of an association rule $Q_1 \Rightarrow Q_2$ in $\mathcal{I}$ is the support of Q_2 in $\mathcal{I}$, an association rule is called* frequent *in $\mathcal{I}$ if Q_2 is frequent in $\mathcal{I}$.*

Definition 3.6. *An association rule $Q_1 \Rightarrow Q_2$ is said to be* ***confident*** *if the support of Q_2 divided by the support of Q_1 exceeds a given minimal confidence threshold.*

Apart from being able to discover interesting relationships among simple conjunctive queries in the form of association rules, our definition of diagonal containment also has the following interesting properties.

Property 3.1. *Given a relational scheme $\mathcal{D}$, the diagonal containment relation $\subseteq^{\Delta}$ defines a pre-order over the set of simple conjunctive queries.*

Proof. It is clear that diagonal containment is reflexive and transitive since both set inclusion and regular containment are. $\square$

Additionally, our support measure is monotone with respect to diagonal containment.

Property 3.2. *Let Q_1 and Q_2 be two simple conjunctive queries. If $Q_1 \subseteq^{\Delta} Q_2$, then support$(Q_1) \leq$ support(Q_2).*

Proof. In the case $\pi(Q_1) = \pi(Q_2)$ this follows directly from the definition of regular containment. In the case that $\pi(Q_1) \subset \pi(Q_2)$, we know that *support*$(Q_1) \leq$ *support*$(\pi_{\pi(Q_1)}Q_2)$. From the definition of support it follows that *support*$(\pi_{\pi(Q_1)}Q_2)$ $\leq$ *support*(Q_2). Transitively the result follows. $\square$

The monotonicity property allows us to prune all simple conjunctive queries contained in an infrequent query. However, in order to use a levelwise Apriori style algorithm, we want to generate queries according to a lattice, as is the case in itemset mining (see Chapter 2). The search space would be a lattice if the diagonal containment relation defined a partial order over the simple conjunctive queries. Unfortunately, this is not the case as the following counterexample demonstrates.

Example 3.4. *Considering $\mathcal{D}$ from Example 3.1, and the following queries:*

$$Q_1 = \pi_A \sigma_{B=C \wedge B=\text{‘}a\text{’}}(R_1 \times R_2)$$
$$Q_2 = \pi_A \sigma_{B=C \wedge C=\text{‘}a\text{’}}(R_1 \times R_2)$$

It is clear that $Q_1 \subseteq^{\Delta} Q_2$ and also $Q_2 \subseteq^{\Delta} Q_1$, since both queries essentially express the same condition. They are equivalent, denoted $Q_1 \equiv^{\Delta} Q_2$. However, as you can clearly see, they are not equal, and hence the antisymmetry requirement of a partial order is not fulfilled.

In generating queries, however, we do not want to consider such equivalent queries since this entails redundant work and redundant output. Therefore we just want to generate one representative of each equivalence class. We call these representative simple conjunctive quires, the *canonical* simple conjunctive queries. If we assume there is a fixed order on the attributes of U, we can define canonical simple conjunctive queries as those queries of which all constraints follow this order. For example a projection on A and B can only be written as $\pi_{AB}Q$, and the equality of A, B and C can only be written as $\sigma_{A=B \wedge A=C}Q$. Furthermore, we also assume that $\bowtie(Q)$ always precedes $\sigma(Q)$, and that constant equality conditions are written using the smallest of the attributes it is equal to. For example if $A = B = \text{‘}a\text{’}$, the corresponding query is written as $\sigma_{A=B \wedge A=\text{‘}a\text{’}}Q$. Furthermore, all constant values must be distinct, e.g. $\sigma_{A=\text{‘}a\text{’} \wedge B=\text{‘}a\text{’}}Q$ is not allowed. Finally we demand $\pi(Q) \cap sch(Q^{\sigma}) = \{\}$. Using this canonical form, we assure that every query can only be represented in one way.

Property 3.3. *Given a relational scheme $\mathcal{D}$, the diagonal containment relation $\subseteq^{\Delta}$ defines a lattice over the set of canonical simple conjunctive queries.*

Proof. It is clear that diagonal containment is a partial order over this set, since if $Q_1 \subseteq^{\Delta} Q_2$ and $Q_2 \subseteq^{\Delta} Q_1$ we know that $Q_1 \equiv^{\Delta} Q_2$. According to the definition of the canonical queries it then follows that $Q_1 = Q_2$, and thus we fulfill the additional antisymmetry requirement. This partial order defines a lattice, since the *join* of two simple conjunctive queries consists of the union of their projected attributes and the intersection of their selections, while the *meet* consists of the intersection of the projected attributes and the conjunction of their selections. The top simple conjunctive query has no selections and projects on all possible attributes, while the bottom simple conjunctive query is simply false. □

We now prove deciding diagonal containment for simple conjunctive queries can be done efficiently.

Lemma 3.1. *If $Q_1 = \pi_{X_1}\sigma_{F_1}$ and $Q_2 = \pi_{X_2}\sigma_{F2}$ are both canonical simple conjunctive queries, $Q_1 \subseteq^{\Delta} Q_2$ if and only if $X_1 \subseteq X_2$ and the set of conditions F_2 is a subset of F_1.*

Proof. It is clear that if F_2 is a subset of F_1 and $X_1 \subseteq X_2$ that $Q_1 \subseteq^{\Delta} Q_2$. Now we prove the reverse is also true. According to the definition of diagonal containment $Q_1 \subseteq^{\Delta} Q_2$ holds if and only if $X_1 \subseteq X_2$ and $Q_1 \subseteq Q_2'$, where we define $Q_2' = \pi_{\pi(Q_1)} Q_2$. This means that the tuples in the result of Q_1 must satisfy the restrictions posed in Q_2'. In a canonical query a condition can only be represented in one way. Thus it follows that all the conditions of set F_2 must be present in F_1. □

Proposition 3.1. *If $Q_1 = \pi_{X_1}\sigma_{F_1}$ and $Q_2 = \pi_{X_2}\sigma_{F2}$ are both canonical simple conjunctive queries of which the projections X_1 and X_2 as well as the sets of conditions F_1 and F_2 are ordered, deciding if $Q_1 \subseteq^{\Delta} Q_2$ takes time linear in the length of Q_1 and Q_2.*

Proof. According to Lemma 3.1, Q_1 is contained in Q_2 if and only if $X_2 \subseteq X_1$ and the set of conditions F_2 is a subset of F_1. Since all sets are ordered, verifying set inclusion takes time linear in the size of the sets. Thus we can conclude that deciding the containment takes time time linear in the size of the queries. □

Note that the assumption that the projections and conditions are ordered is realistic, since we are only dealing with queries we generate ourselves.

3.1.2 Cartesian Products

Since we want to obtain a concise set of interpretable patterns, it is of no use to consider simple conjunctive queries that essentially represent a cartesian product, as the following example illustrates.

Example 3.5. *Consider the following simple conjunctive queries.*

$$\begin{aligned} Q_1 &: \pi_{R_1.A, R_2.B}(R_1 \times R_2) \\ Q_2 &: \pi_{R_1.A}\sigma_{R_2.B=\text{‘c’}}(R_1 \times R_2) \end{aligned}$$

As can be seen, Q_1 is a simple cartesian product of two attributes from different relations, and hence, its support is simply the product of the number of unique values of the two attributes. Now, whenever a simple conjunctive query is not frequent, there most probably exist also several other versions of that query including a cartesian product with another attribute which contains enough tuples to make the product exceed the minimum support threshold.

Similarly, almost every frequent query will be duplicated many times due to cartesian products. This case is illustrated by Q_2. Here, the output equals $\pi_{R_1.A}$ only if there exists a tuple in R_2 containing the value ‘c’ for attribute B. Obviously, almost every frequent query could be combined like that with every possible value in the database.

In order not to consider these types of queries we add an additional constraint. According to the definition, a simple conjunctive query Q has a join, $\bowtie(Q)$, expressed using a conjunction of selection conditions of the form $R_i.A = R_j.A'$. Such a conjunctive condition induces a partition of U, where every block β of this partition is a maximal set of attributes such that for all $R_i.A$ and $R_j.A'$ in β, $R_i.A = R_j.A'$ is a consequence of $\bowtie(Q)$. We denote this partition by $blocks(\bowtie(Q))$ and we say that R_i and R_j are *connected through* $\bowtie(Q)$. To simplify notation, given a simple conjunctive query Q, the corresponding partition of U, $blocks(\bowtie(Q))$ is simply denoted by $blocks(Q)$.

Example 3.6. *Considering the schema $\mathcal{D}$ where $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, the query $Q = \pi_B\sigma_{A=C\wedge C=D}(R_1 \times R_2)$ induces the partition $blocks(Q) = \{\{A, C, D\}, \{B\}, \{E\}\}$. In this case the relation R_1 and R_2 are connected through $\bowtie(Q)$.*

We only consider those simple conjunctive queries $Q = \pi_X\sigma_F(R_1 \times \cdots \times R_n)$ where all relation names occurring in X or in $\sigma(Q)$ are *connected* through $\bowtie(Q)$, since all other conjunctive queries represent cartesian products, as illustrated in the following example.

Example 3.7. *Considering the schema $\mathcal{D}$ from Example 3.6, the query $Q = \pi_{AD}\sigma_{(A=B)\wedge(E=e)}(R_1 \times R_2)$ is* not *a considered simple conjunctive query because R_1 and R_2 are not connected through the condition $A = B$. Computing the answer to this query requires to explicitly consider the cartesian product $R_1 \times R_2$.*

On the other hand, $Q_1 = \pi_{AD}\sigma_{(A=C)\wedge(E=e)}(R_1 \times R_2)$ is a considered simple conjunctive query such that $\bowtie(Q_1) = (A = C)$, $\pi(Q_1) = AD$, $\sigma(Q) = (E = e)$ and $Q^\sigma = e$. $blocks(Q_1)$ contains four blocks, namely: $\{A, C\}$, $\{B\}$, $\{D\}$ and $\{E\}$. Notice that, in this case, computing the answer to Q_1 does not require to explicitly consider the cartesian product $R_1 \times R_2$, since R_1 and R_2 are joined through $A = C$.

We are able to avoid generating conjunctive queries with disconnected relations by using the graph theoretic concept of connected components, which we expand on in Section 3.2.1.

3.1.3 Basic Association Rules

Given the association rules $Q_1 \Rightarrow Q_2$ and $Q_2 \Rightarrow Q_3$, due to the definition of confidence it holds that confidence$(Q_1 \Rightarrow Q_3)$ is equal to confidence$(Q_1 \Rightarrow Q_2) \times$ confidence$(Q_2 \Rightarrow Q_3)$. Consequently, we only compute confident rules $Q_1 \Rightarrow Q_2$ such that there exists no Q in such that $Q_1 \Rightarrow Q$ and $Q \Rightarrow Q_2$ are association rules. These association rules, which we call *basic rules*, are characterised according to Proposition 3.2, defined below.

Proposition 3.2. *An association rule $Q_1 \Rightarrow Q_2$ is a* ***basic rule*** *if and only if it satisfies one of the following:*

1. *$\pi(Q_2) \subset \pi(Q_1)$ and $\sigma(Q_1) = \sigma(Q_2)$ and $\bowtie(Q_2) = \bowtie(Q_1)$, and there does not exist a schema X such that $\pi(Q_2) \subset X \subset \pi(Q_1)$.*

2. *$\pi(Q_1) = \pi(Q_2)$ and $\sigma(Q_1) \subset \sigma(Q_2)$, and $\bowtie(Q_1) = \bowtie(Q_2)$, and there does not exist a set of conditions Y of the form $R_k.A = c$ such that $\sigma(Q_1) \subset Y \subset \sigma(Q_2)$.*

3. *$\pi(Q_2) = \pi(Q_1)$ and $\sigma(Q_1) = \sigma(Q_2)$ and $\bowtie(Q_2) \subset \bowtie(Q_1)$, and there does not exists a set of conditions J of the form $R_i.A = R_j.A'$ such that $\bowtie(Q_2) \subset J \subset \bowtie(Q_1)$.*

Proof. It can easily be verified that if a rule satisfies one of these cases that it is a basic rule. Now we prove the other direction. Let us denote $X \subset^1 Y$ if and only if $X \subset Y$ and there does not exist a Z such that $X \subset Z \subset Y$.

A rule $Q_1 \Rightarrow Q_2$ holds if and only if $Q_2 \subseteq^\Delta Q_1$. We have shown in Lemma 3.1 that then $X_2 \subseteq X_1$ and $F_1 \subseteq F_2$. $Q_1 \Rightarrow Q_2$ is a basic rule, thus there does not exist a Q such that $Q_1 \Rightarrow Q$ and $Q \Rightarrow Q_2$. This means that there does not exist a Q with projection X and conditions F such that (1) $X_2 \subseteq X \subseteq X_1$ and (2) $F_1 \subseteq F \subseteq F_2$. For (1) to hold it must be that (1.1) $X_2 \subset^1 X_1$ or (1.2) $X_2 = X_1$ holds. Similarly for (2) it must be that (2.1) $F_1 \subset^1 F_2$ or (2.2) $F_1 = F_2$. If (1.1) holds, then (2.1) cannot hold since otherwise a Q exists, namely the Q such that $X = X_1$ and $F = F_2$, thus (1.1) must hold, resulting in case 1. If (1.2) holds, (2.2) cannot hold since otherwise $Q_1 = Q_2$ which we do not consider as an association rule. Thus (2.1) must hold. $F_1 \subset^1 F_2$ can be split up in two cases since $F_i = \sigma(Q_i) \wedge \bowtie(Q_1)$. It must hold that $\bowtie(Q_1) = \bowtie(Q_2)$, and $\sigma(Q_1) \subset^1 \sigma(Q_2)$ or $\sigma(Q_1) = \sigma(Q_2)$ and $\bowtie(Q_2) \subset^1 \bowtie(Q_1)$, which are cases 2 and 3. □

Generating all canonical queries, covers the complete set of possible queries since for each query there is always an equivalent canonical query. However, generating all rules over all canonical queries is not enough to cover all rules over all queries.

Example 3.8. *Considering the schema $\mathcal{D}$ where $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, the valid association rule*

$$\pi_B(R_1 \times R_2) \Rightarrow \pi_B \sigma_{B=C}(R_1 \times R_2)$$

cannot be represented using canonical queries. Since this would require the following rule

$$\pi_B(R_1 \times R_2) \Rightarrow \pi_{BC} \sigma_{B=C}(R_1 \times R_2)$$

which is not valid since it is clear that $\pi_{BC}\sigma_{B=C}(R_1 \times R_2) \not\subseteq^{\Delta} \pi_B(R_1 \times R_2)$ *because of the fact that* $\{B, C\} \not\subseteq \{B\}$

Therefore, when generating association rules we do take equivalent queries into account. This allows us to use Proposition 3.2 to generate all confident basic association rules, starting from canonical queries. This is discussed in detail in Section 3.2.3 where rule generation is explicated.

3.1.4 Problem Statement

Having defined the preliminaries, we can now formalise our main problem statement as follows:

Given a database with a scheme $\mathcal{D}$, find all frequent and confident basic association rules over, cartesian product free, simple conjunctive queries with a given minimum support threshold and a minimum confidence threshold.

3.2 Algorithm: Conqueror

In order to find all confident basic association rules, we propose the algorithm **Conqueror** (**Con**junctive **Quer**y Generat**or**). It is a levelwise, Apriori style algorithm (see Chapter 2). It is divided into two phases. In a first phase, all frequent simple conjunctive queries are generated. Then, in a second phase, all confident association rules over these frequent queries are generated. The second phase is considerably easier, as most computationally intensive work and data access is performed in the first phase. After all, for every generated simple conjunctive query, we need to compute its support in the database, while association rule generation merely needs to find couples of previously generated queries and compute their confidence. Consequently, most of our attention goes to the first phase, starting with the candidate generation procedure.

3.2.1 Candidate Generation

In the candidate generation phase, we essentially generate all possible instantiations of X and F in $\pi_X\sigma_F(R_1 \times \cdots \times R_n)$. We make sure we generate every query only once, and also take care not to generate the undesirable cartesian product queries.

The Conqueror algorithm is made up of three loops:

Join loop: Generate all instantiations of F, without constants, *i.e.* $\bowtie(Q)$, in a breadth-first manner.

Projection loop: For each generated join $\bowtie(Q)$, generate all instantiations of X, *i.e.* $\pi(Q)$, in a breadth-first manner, and test their frequency against the given instance $\mathcal{I}$.

Selection loop: For each generated projection-join query, add constant assignments to F, *i.e.* $\sigma(Q)$, in a breadth-first manner.

In order to make sure candidate queries are generated at most once, we assume that all attributes are ordered according to a fixed ordering. This ordering is implicit in lines 1 and 10 in Algorithm 3.1 (in the sense that the k-th element in the string refers to the k-th attribute according to the ordering), and is explicitly used in line 10 in Algorithm 3.3 and line 5 in Algorithm 3.4. Note that this also implies the containment can be checked in linear time as stated in Section 3.1.1

In the remainder of this section we describe each of these loops in detail. In Section 3.2.2 we describe our implemented techniques to efficiently evaluate each query on the database.

Join loop

Recall that for a simple conjunctive query Q, the conditions F without constants consist of a conjunction of equalities between attributes, and thus essentially represent a join (therefore we denote this set of conditions as $\bowtie(Q)$). As stated in Section 3.1.2, $\bowtie(Q)$ induces a partition of the set of all attributes U, denoted *blocks*(Q) where attributes are in the same block of the partition if they are equal according to the set of equality conditions $\bowtie(Q)$. Generating all joins is therefore reduced to generating all partitions of U. Generating all partitions of a set is a well studied problem for which efficient solutions exist. We use the so called *restricted growth string* for generating all partitions [Weisstein, 2009].

Assuming a fixed order over a set of attributes, a restricted growth string is an array $a[1 \dots m]$ where m is the total number of attributes, and $a[i]$ is the block identifier of the block in the partition in which attribute i occurs. Obviously, a partition can be represented by several such strings, but in order to identify a unique string for each partition, the so called *restricted growth* string satisfies the following growth inequality (for $i = 1, 2, \dots, n-1$, and with $a[1] = 1$):

$$a[i+1] \leq 1 + \max a[1], a[2], \dots, a[i] \tag{3.1}$$

Example 3.9. *Considering the schema $\mathcal{D}$ where $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, the set U of all attributes occurring in $\mathcal{D}$ is $\{A, B, C, D, E\}$. Then, the restricted growth string* 12231 *represents the condition $(A = E) \wedge (B = C)$, which corresponds to the partition $\{\{A, E\}, \{B, C\}, \{D\}\}$.*

Algorithm 3.1 Conqueror

```
Input: Database $\mathcal{D}$, threshold minsup, threshold minconf, most specific join msj
Output: $\mathcal{R}$ the set of confident association rules
 1: $\bowtie(Q)$ := “1” //initial restricted growth string
 2: push(Queue, Q)
    //Join Loop
 3: while not Queue is empty do
 4:    JQ := pop(Queue)
 5:    if rgs does not represent a cartesian product then
 6:       $\mathcal{F} := \mathcal{F} \cup$ ProjectionLoop(JQ)
 7:    children := RestictedGrowth($\bowtie$(JQ), m)
 8:    for all rgs in children do
 9:       if join defined by rgs is not more specific than msj then
10:          $\bowtie$(JQC) := rgs
11:          push(Queue, JQC)
12: $\mathcal{R}$ := RuleGeneration($\mathcal{F}$, minconf)
13: return $\mathcal{R}$
```

Notice that this restriction corresponds to the requirement we have for canonical conjunctive queries. The algorithm to generate all join queries is shown in Algorithm 3.1. In order to efficiently generate all partitions without generating duplicates, we initially start with the singleton string “1”, representing the first attribute belonging to block 1, and all remaining attributes belong to their own unique block (thus essentially the restricted growth string “123. . .m”, although we do not explicitly represent it this way).

Then, given such a string representing a specific partition, all more specific partitions are generated making use of the restricted growth principle. This is shown in Algorithm 3.2. Essentially we generate the children of a partition, by adding one of the remaining attributes to an existing block. To make sure no duplicates are generated, we do not add an attribute to an existing block if any of the attributes coming after that have already been assigned to an existing block, *i.e.*, we make sure the growth inequality 3.1 holds. This traversal of the search space for four attributes is illustrated in Figure 3.1. Since we are using a queue and only generating the children, essentially, our algorithm performs a breadth-first traversal over this tree.

Before generating all possible projections for a given join, we first determine whether the selection represents a cartesian product (line 5, Algorithm 3.1). If so, we skip generating projections for this join and continue the loop, joining more attributes until the join no longer represents a cartesian product.

Intuitively, to determine whether a join represents a cartesian product, we

Algorithm 3.2 Restricted Growth

Input: String *prefix*, Length m

Output: *list* of restricted growth strings with prefix *prefix* and length m

```
1: $list \leftarrow \{\}$
2: $last \leftarrow \text{length}(prefix)$
3: if $last < m$ then
4:     for $i = last$ to $m - 1$ do
5:         $max \leftarrow \max(\{prefix[j] \mid 0 \leq j < last\})$
6:         $nprefix \leftarrow prefix$
7:         if $i > last$ then
8:             for $k = last$ to $i - 1$ do
9:                 $max \leftarrow max + 1$
10:                $nprefix[k] \leftarrow max$
11:        for $l = 1$ to $max$ do
12:            $nprefix[i] := l$
13:            add($list$, $nprefix$)
14: return $list$
```

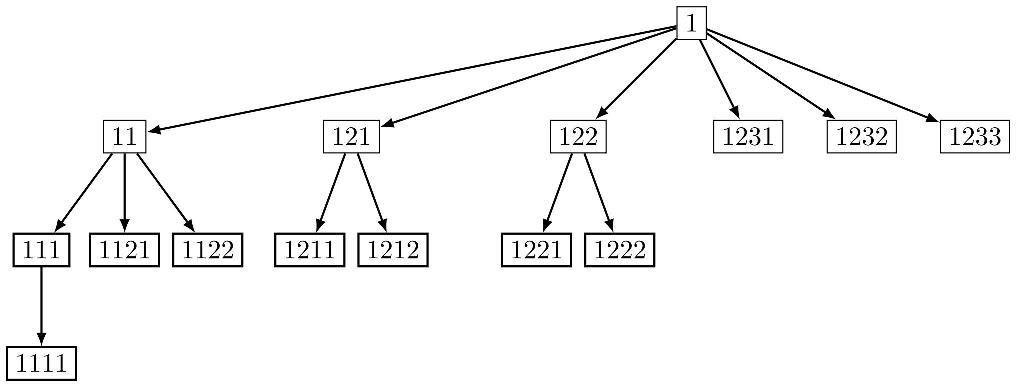
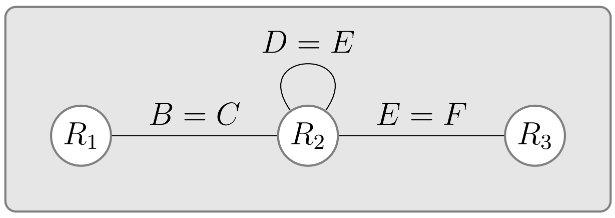
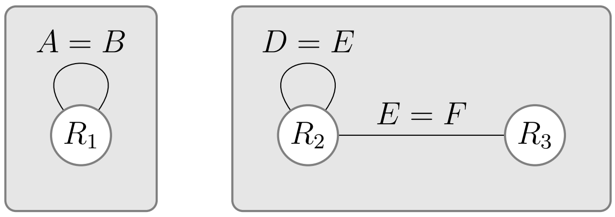


Figure 3.1: Restricted Growth Expansion



(a) a single connected component



(b) disconnected components

Figure 3.2: Connected and Disconnected Components

interpret each join as an undirected graph. As stated in Section 3.1.2, relations can be *connected through* a join condition $\bowtie(Q)$, essentially if some of their attributes get joined. Consider the graph of a join $\bowtie(Q)$ to consist of a node for each relation, and a connection between these nodes if they are connected through $\bowtie(Q)$. Then, we only allow those joins for which all edges are in the same single connected component. All other joins represent cartesian products, and consequently, we prune these in the join loop (line 5, Algorithm 3.1).

Example 3.10. *Given the scheme $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$, $sch(R_2) = \{C, D, E\}$ and $sch(R_3) = \{F\}$, the join*

$$R_1.B = R_2.C \wedge R_2.E = R_3.F \wedge R_2.D = R_2.E,$$

represented in Figure 3.2a, results in a single connected component, while the join

$$R_1.A = R_1.B \wedge R_2.E = R_3.F \wedge R_2.D = R_2.E,$$

represented in Figure 3.2b, results is two disconnected components. Hence, independent of the projected attributes, any simple conjunctive query using the second join always represents a cartesian product.

In practice, it does not make any sense to compare attributes of incomparable types. Moreover, the user might see no use in comparing addresses with names even though both could be typed as strings. Therefore, we allow the user to specify the sensible joins in the database by providing the *most specific join* to be

Algorithm 3.3 Projection Loop

```
Input: Conjunctive Query $Q$
Output: $\mathcal{F}_Q$ set of frequent queries with the join of $Q$
 1: $\pi(Q) :=$ blocks$(Q)$ //all connected blocks of $\bowtie(Q)$
 2: push($Queue$, $Q$)
 3: while not $Queue$ is empty do
 4:    PQ := pop($Queue$)
 5:    if monotonicity(PQ) then //check monotonicity
 6:       $support$(PQ) := EvaluateSupport(PQ)
 7:       if $support$(PQ) $>$ $minsup$ then
 8:          $\mathcal{F}_Q := \mathcal{F}_Q \cup$ SelectionLoop(PQ)
 9:          $removed$ := blocks(PQ) $\notin \pi$(PQ)
10:          $torem$ := blocks(PQ) $>$ last of $removed$ //order on blocks is supposed in order
             to generate uniquely
11:          for all $p_i \in torem$ do
12:             $\pi$(PQC) $\leftarrow \pi$(PQ) with block $p_i$ removed
13:             push($Queue$,PQC)
14: return $\mathcal{F}_Q$
```

considered. That is, a partition of all attributes, such that only the attributes in the same block are allowed to be compared to each other. When generating a new partition, it is compared to the most specific join (line 9, , Algorithm 3.1), and only considered if it is more general or equal. By default, if no most specific join is specified, every possible join of every attribute pair is considered.

Projection loop

In this loop, for each join, all projections are generated. However, we only consider those projections consisting of those attributes whose relations are part of the single connected component (as determined by the cartesian product test in Algoritm 3.1). Indeed, we recall from Section 3.1.2 that projecting on any other attribute would result in a cartesian product.

Example 3.11. *Considering the scheme from Example 3.10, the join $R_1.B = R_2.C$ represents one connected component and therefore does not represent a cartesian product. However, the query $\pi_{A,F}\sigma_{B=C}(R_1 \times R_2 \times R_3)$ does represent a cartesian product, namely that of all A values for which $B = C$ with all F values in R_3.*

Therefore, our algorithm (Algorithm 3.3) starts with the set of all allowed attributes for the given join (namely, those of the relations present in the connected component), and then generates subsets in a breadth-first manner with respect to diagonal containment. But even then, not all subsets are generated, as this might result in duplicate non-canonical queries.

Example 3.12. *Still considering $\mathcal{D}$ from Example 3.10, it is clear that the queries*

$$\pi_{B,C}\sigma_{B=C}(R_1 \times R_2 \times R_3), \pi_B\sigma_{B=C}(R_1 \times R_2 \times R_3) \text{ and } \pi_C\sigma_{B=C}(R_1 \times R_2 \times R_3)$$

are all equivalent.

Indeed, when we remove an attribute from the projection while there might still be other attributes from its block in the join partition, we obtain an equivalent query. Therefore, we only consider simultaneous removal of all attributes from a single block of $blocks(Q)$, *i.e.* we are considering block-sets instead of attribute-sets. Note that in order to avoid generating cartesian products as stated above, the function blocks(Q) in the algorithms returns the set of *connected* blocks of a restricted growth string, that is, blocks(Q) returns the connected part of the partition $blocks(Q)$ (line 1, Algorithm 3.3).

For every generated projection, we perform a monotonicity check (line 5). We expand upon monotonicity in Section 3.2.3. If the query passes, it is evaluated against the database (line 6). If the query turns out to be infrequent, none of its sub-projections or selections are considered anymore, as they must be infrequent too. If the query is frequent we start the selection loop (line 8) and generate its sub-projections as described above.

Selection loop

The last part we still need to consider in query generation is those selection conditions of the type $R_k.A = `c'$, *i.e.* constant equalities. For a simple conjunctive query Q, we denote this subset of F as $\sigma(Q)$. As was the case in the projection loop, we must take care not to generate equivalent non-canonical queries.

Example 3.13. *Considering $\mathcal{D}$ from Example 3.10, it is clear that the queries*

$$\pi_A\sigma_{B=C\wedge C=`a'}(R_1 \times R_2 \times R_3) \text{ and } \pi_A\sigma_{B=C\wedge B=`a'}(R_1 \times R_2 \times R_3)$$

are equivalent with respect to support.

In order to only generate a certain constant equality once, we consider constant equality as a property of the connected blocks of the partition $blocks(Q)$. In Example 3.13 the connected blocks are $\{\{A\},\{B,C\},\{D\},\{E\}\}$, and thus the constant equality should be generated as $\{B,C\} = `a'$. Therefore, generating the candidates for constant equalities comes down to generating all subsets of blocks(Q).

Algorithm 3.4 Selection Loop

```
Input: Conjunctive Query $Q$
Output: $\mathcal{F}_Q$ set of frequent queries with the join and projection of $Q$
 1: push($Queue$,$Q$)
 2: while not $Queue$ is empty do
 3:    CQ := pop($Queue$)
 4:    if $\sigma$(CQ) = $\emptyset$ then
 5:       $toadd$ := all blocks of blocks(CQ) $\notin \pi(CQ)$
 6:    else
 7:       $uneq$ := all blocks of blocks(CQ) $\notin$ ($\sigma$(CQ) $\cup$ $\pi$(CQ))
 8:       $toadd$ $\leftarrow$ blocks in $uneq$ > last of $\sigma$(CQ) //order on blocks is supposed
 9:    for all $B_i \in toadd$ do
10:       $\sigma$(CQC) := $\sigma$(CQ) $\cup$ $B_i$
11:       if exist frequent constant values for $\sigma$(CQC) in the database then
12:          $\mathcal{F}_Q$ := $F_Q$ $\cup$ CQC
13:          push($Queue$,CQC)
14: return $\mathcal{F}_Q$
```

This is done in a level-wise, breadth-first manner as shown in Algoritm 3.4. First, we assign a constant to all single block in blocks(Q) (line 5). On the next level, we assign constants to two different blocks, only if these constants already resulted in frequent queries separately. This is repeated until no more combinations can be generated (line 8). Again, there is one exception. We do not allow constants to be assigned to blocks that are in the projection (line 5 and 8). Indeed, also these would be equivalent to queries in which this block is not in the projection, as the following example demonstrates:

Example 3.14. *Considering $\mathcal{D}$ from Example 3.10, it is clear that the queries*

$$\pi_A\sigma_{B=C\wedge C=\text{`}a\text{'}}(R_1 \times R_2 \times R_3) \text{ and } \pi_{AC}\sigma_{B=C\wedge C=\text{`}a\text{'}}(R_1 \times R_2 \times R_3)$$

are equivalent with respect to support.

For every generated equality of constants, the resulting query is evaluated against the database (line 11). If the query turns out to be infrequent, then no more specific constant assignments are generated, as they must be infrequent too. The matter of specific constant *values* is handled in the database. This, together with query evaluation, is discussed in the next section.

3.2.2 Candidate Evaluation

In order to get the support of each generated query, Conqueror evaluates them against the relational database by translating each query into SQL. In general, in order to get the support of a general conjunctive query

$$\pi_X \sigma_F (R_1 \times \cdots \times R_n)$$

we can translate it to the following SQL query, and evaluate it against the relational database:

```
SELECT COUNT(DISTINCT $X$)
FROM $R_1, \cdots, R_n$
WHERE $F$
```

It is clear that the algorithm requires many queries with the same join condition to be evaluated, *i.e.* in the projection and selection loop. For efficiency reasons, we therefore materialise these joins and rewrite the more specific queries to use it. Thus, the above translation is only performed for queries with the most general projection and no constant equalities. The result of such a query is then stored in a temporary table (τ). We can now rewrite more specific queries to use these temporary tables resulting in a more efficient evaluation as we are now querying just a specific part of the database and we no longer have to perform a possibly expensive join operation for each more specific query. All more specific projections X', having the same join condition, are evaluated by the relational algebra query $\pi_{X'}\tau$ or in SQL:

```
SELECT COUNT(DISTINCT $X'$)
FROM $\tau$
```

To evaluate the more specific simple conjunctive queries containing constant equation we developed some additional optimisations. It is hard and inefficient to retrieve the potentially large number of constant values from the database and then keep them in main memory. Therefore we also use temporary tables to store them in the database itself. Creating the SQL queries for these simple conjunctive queries involves combining the various temporary tables from previous loops using the monotonicity property, as we now explain. At the first level we generate more specific queries that contain only one constant equation. These are queries of the form $\pi_{X'}\sigma_{\bowtie(Q)\wedge A=?}(R_1 \times \cdots \times R_n)$. Since these queries are also more specific than a previously materialised join (of query Q, supposedly in table τ), we can rewrite them as $\pi_{X'}\sigma_{A=?}\tau$. To evaluate these in the database queries are written in SQL as follows:

```
SELECT A, COUNT(*) AS sup
FROM τ
GROUP BY A
HAVING COUNT(*) >= minsup
```

The result of such a query is stored in a new temporary table (τ_A). It holds all the constant values v for A together with their support, such that the query $\pi_{X'}\sigma_{A=v}\tau$ satisfies the minimal support requirement. Essentially we are performing the evaluation of many different simple conjunctive queries (all with different values for v) at once and immediately pruning the infrequent ones, by using just a single SQL query.

On the following levels, these temporary tables are combined to generate queries containing more than one constant equation as illustrated in the examples below.

Example 3.15. *Let τ_A and τ_B be the temporary tables holding the constant values for the attributes A and B together with their support (generated in the previous level). We can now generate the SQL for the query $\pi_{X'}\sigma_{A=?\wedge B=?}\tau$ as follows:*

```
SELECT A, B, COUNT(*) FROM
      (SELECT A,B, X' FROM
            τ NATURAL JOIN
                  (SELECT * FROM
                        (SELECT A FROM τ_A)
                        NATURAL JOIN
                        (SELECT B FROM τ_B)
                  )
      )
GROUP BY A,B
HAVING COUNT(*) >= minsup
```

In this case we are using the values already obtained for A and B in generating the combinations, and using a join with the temporary table τ to evaluate against the database, immediately using the minimal support value minsup to only get frequent queries. Essentially we are making use of the monotonicity property, similar to the generation of itemsets from subsets (Chapter 2). The result of this query is also stored in a temporary table $\tau_{A,B}$, and will be exploited in following levels as we see next.

In the previous example the join actually was a simple product of the frequent A and B values, but as one advances more levels (*i.e.*, more blocks are equal to constants) it becomes a real join on the common attributes as illustrated in the example below.

Example 3.16. *This is the generated SQL for the query* $\pi_{X'}\sigma_{A=?\wedge B=?\wedge C=?}\tau$. *It uses the temporary tables* τ, $\tau_{A,B}$, $\tau_{A,c}$, $\tau_{B,C}$.

```
SELECT $A, B, C$, COUNT(*) FROM
       (SELECT $A, B, C, X'$ FROM
              $\tau$ NATURAL JOIN
                     (SELECT * FROM
                            (SELECT $A, B$ FROM $\tau_{A,B}$)
                            NATURAL JOIN
                            (SELECT $A, C$ FROM $\tau_{A,C}$)
                            NATURAL JOIN
                            (SELECT $B, C$ FROM $\tau_{B,C}$)
                     )
       )
GROUP BY $A, B, C$
HAVING COUNT(*) >= $minsup$
```

This SQL based approach, allows us to efficiently use the features of the RDBMS to handle the potentially sizable numbers of constant values.

3.2.3 Monotonicity

Candidate evaluation is costly, especially in our case since it involves sending an SQL query to a relational database and retrieving the result. In order to reduce the input/output cost we want to avoid evaluating queries as much as possible. Luckily we know from Section 3.1.1 that diagonal containment satisfies the monotonicity property with respect to support. In the algorithm pseudocode references are made to a monotonicity function which is called before the evaluation of the support of a candidate query in the database. Due to the monotonicity of support we only check those queries Q' such that $Q \subset^{\Delta} Q'$ (where $Q \subset^{\Delta} Q'$ if $Q \subseteq^{\Delta} Q'$ but not $Q \equiv^{\Delta} Q'$) and there does not exist a Q'' such that $Q' \subset^{\Delta} Q'' \subset^{\Delta} Q$. We denote this as $Q {\subset^{\Delta}}^{1} Q'$ and say that Q is directly contained in Q'. The monotonicity function therefore checks if all Q' where $Q {\subset^{\Delta}}^{1} Q'$ are frequent. If not, we prune the candidate Q. We also showed in Section 3.1.1 that checking diagonal containment can be done efficiently. Based on this we also generate all directly contained queries efficiently. Essentially it comes down to generating all queries with one less restriction than the considered candidate query Q, since $Q {\subseteq^{\Delta}}^{1} Q'$ if and only

Algorithm 3.5 Monotonicity

```
Input: Conjunctive Query $Q$
 1: for all blocks $\beta$ in $blocks(Q) \notin \pi(Q)$ do
 2:   $PP := PP \cup \{Q'\}$ where $\pi(Q') = \pi(Q) \cup \beta$, $\bowtie(Q') = \bowtie(Q)$ and $\sigma(Q') = \sigma(Q)$
 3: for all blocks $\beta \in \sigma(Q)$ do
 4:   $SP := SP \cup \{Q'\}$ where $\sigma(Q') = \pi(Q) \backslash \beta$, $\bowtie(Q') = \bowtie(Q)$ and $\pi(Q') = \pi(Q)$
 5: for all blocks $\beta$ in $blocks(Q)$ do
 6:   for all splits of $\beta$ in $\beta_1$ and $\beta_2$ do
 7:     $\bowtie(Q') = (\bowtie(Q) \setminus \beta) \cup \{\beta_1, \beta_2\}$
 8:     if $\beta \in \pi(Q)$ then
 9:       $\sigma(Q') = \sigma(Q)$
10:       $JP := JP \cup \{Q'\}$ where $\pi(Q') = (\pi(Q) \setminus \beta) \cup \beta_1$
11:       $JP := JP \cup \{Q'\}$ where $\pi(Q') = (\pi(Q) \setminus \beta) \cup \beta_2$
12:       $JP := JP \cup \{Q'\}$ where $\pi(Q') = \pi(Q)$
13:     else
14:       $\pi(Q') = \pi(Q)$
15:       if $\beta \in \sigma(Q)$ then
16:         $JP := JP \cup \{Q'\}$ where $\sigma(Q') = (\sigma(Q) \setminus \beta) \cup \beta_1$
17:         $JP := JP \cup \{Q'\}$ where $\sigma(Q') = (\sigma(Q) \setminus \beta) \cup \beta_2$
18:       else
19:         $JP := JP \cup \{Q'\}$ where $\sigma(Q') = \sigma(Q)$
20: for all $MGQ$ in $(JP \cup PP \cup SP)$ do
21:   if $MGQ$ is not a cartesian product then
22:     if $support(MGQ) < minsup$ then
23:       return false
24:     R($Q$) := R($Q$) ∪ {MGQ} //lhs for potential rule
25: return true
```

if $Q' \Rightarrow Q$ is a basic rule, and thus Proposition 3.2 applies. This process is given in Algorithm 3.5.

A simple conjunctive query consists of a join, a projection and constant equalities, and each of these parts represents a collection of restrictions. Since we represent simple conjunctive queries in a canonical way, removing one restriction from either of these parts generates a query with one restriction less. Doing this for the projection is straightforward. As the most general projection is known, one can easily add a block of attributes that is not in the projection. This gives us a query with one less restriction, and doing this for all blocks not in the projection results in the generation of all queries with one restriction less pertaining to the projection (line 1). For the constant equalities the reverse is true, instead of adding blocks of constant equations we have to remove blocks to become more general, but this can easily be done (line 3). The join of a query is represented by the partition into blocks of attributes. To generate the queries with one less restriction with respect to the join, we need to remove 'joins'. This comes down to splitting blocks of the partition into two parts. In order to generate all of them we have to split every block in every possible way one at a time (line 6). One difficulty here lies in the adaptation of the projection and constants to this new situation. A split block could have been projected. This then results in three possible parents: both new blocks are projected, only the first block is projected or only the second block is projected (line 8). Because blocks of the partition can also have constants associated with them, splitting these blocks also means that this constant equation has to be split. This results in two more general queries, one for each part of the split with the constants associated with it (line 15). We must note that some of these generated parents can represent cartesian products. As we opted to not consider cartesian products we have no support values for these queries and thus are unable to check if monotonicity holds in that case. The simple solution we adopt is to simple ignore these cartesian products (line 21), essentially assuming that they are frequent. This does imply that the monotonicity check is not complete.

3.2.4 Association Rule Generation

The generation of basic association rules is performed in a straightforward manner. For all queries Q_1 the algorithm needs to find all queries Q_2 such that Q_2 is directly contained in Q_1. Since these are exactly those queries that are generated in the monotonicity check, we integrate rule generation there (Algorithm 3.5, line 24). Note that these are only potential rules. Only queries that pass the monotonicity check and that are found frequent after evaluation in the database can generate confident association rules. The confident association rule generation is shown in Algorithm 3.6. Here we simply compute the confidence $(\mathit{support}(Q)/\mathit{support}(MGQ))$. If found to be confident the rule $Q \Rightarrow \pi_{\pi(Q)}MGQ$ can be added to the rule output of the algorithm. Notice that we project MGQ onto the projection of Q as it is possible that MGQ has a larger projection due to its canonical form. As we explained in 3.1.1, we want to cover all rules over all queries, instead of just all rules of the set of canonical queries. The projection creates an equivalent query and a valid association rule. As stated, the time spent in this whole process is negligible compared to the query generation.

3.3 Conqueror Experimental Results

We performed several experiments using our prototype on the backend database of an online quiz website [Bocklandt, 2008] and a snapshot of the Internet Movie

Algorithm 3.6 Rule Generation

Input: set of frequent queries FQ, threshold *minconf*

Output: set of confident association rules CR

```
1: for all $Q \in$ FQ do
2:    for all MGQ $\in$ R($Q$) do
3:       if $(support(Q)/support(MGQ)) \geq minconf$ then
4:          CR := CR $\cup \{Q \Rightarrow \pi_{\pi(Q)} MGQ\}$
5: return CR
```

Database (IMDB) [IMDB, 2008]. The quiz database (QuizDB) consists of two relations named *score* and *quizzes*, containing respectively 866 755 tuples and 4884 tuples. The IMDB snapshot contains the relations *actors* (45 342 tuples), *movies* (71 912 tuples) and *genres* (21 tuples) connected with the relations *actormovies* and *genremovies*. A summary of the characteristics of these databases can be found in Table 3.3a and Table 3.3b respectively.

The experiments were performed on a standard computer with 2GB RAM and a 2.16 GHz processor. The prototype algorithm was written in Java using JDBC to communicate with an SQLite 3.4.0 relational database[1].

What follows are some examples of interesting patterns discovered by our algorithm. Note that we abbreviate the relation names to improve readability.

3.3.1 Movie Database

The IMDB snapshot consist of three tables *actors (a)*, *movies (m)* and *genres (g)*, and two tables that represent the connections between them namely *actormovies (am)* and *genremovies (gm)*. First of all let us take a look at some patterns that are generated as a result of vertical containment.

Example 3.17. *The following rule, has 100% confidence and can be interpreted as a* ***functional dependency****:*

$$\pi_{\text{mid,name}}(\text{movies}) \Rightarrow \pi_{\text{mid}}(\text{movies}).$$

Interestingly enough the rule

$$\pi_{\text{mid,name}}(\text{movies}) \Rightarrow \pi_{\text{name}}(\text{movies})$$

has 99.99% confidence, so we can derive that in our database snapshot there are different movies which have the same name (although not a lot). This type of association rules could be considered as a certain kind of ***approximate functional dependency*** *[Kivinen & Mannila, 1995]. Such approximate dependencies*

[1]The source code of Conqueror can be downloaded at `http://www.adrem.ua.ac.be`.

attribute	#values
actors.*	45342
actors.aid	45342
actors.name	45342
genres.*	21
genres.gid	21
genres.name	21
movies.*	71912
movies.mid	71912
movies.name	71906
actormovies.*	158441
actormovies.aid	45342
actormovies.mid	54587
genremovies.*	127115
genremovies.gid	21
genremovies.mid	71912

(a) IMDB

attribute	#values
scores.*	868755
scores.score	14
scores.player	31934
scores.qid	5144
scores.date	862769
scores.results	248331
scores.month	12
scores.year	6
quizzes.*	4884
quizzes.qid	4884
quizzes.title	4674
quizzes.author	328
quizzes.category	18
quizzes.language	2
quizzes.number	539
quizzes.average	4796

(b) QuizDB

Figure 3.3: Number of tuples per attribute in the QuizDB and IMDB databases

are considered very interesting pattern types in the context of database cleanup, since we might expect that this dependency should hold. We expand on approximate dependencies in Section 3.4.

Now let us investigate some examples that illustrate regular containment.

Example 3.18. *We can conclude that every movie has a genre because of the following association rule with 100% confidence*

$$\pi_{\text{m.mid}}(\text{movies}) \Rightarrow \pi_{\text{m.mid}}\sigma_{\text{gm.mid=m.mid}}(\text{movies} \times \text{genremovies}).$$

It is easy to see that this type of rules describes ***inclusion dependencies*** *occurring in the database, a dependency type we will also discuss in Section 3.4. On the contrary, in our database, not every movie has to have an actor associated with it as the following rule only has 76% confidence*

$$\pi_{\text{m.mid}}(\text{movies}) \Rightarrow \pi_{\text{m.mid}}\sigma_{\text{am.mid=m.mid}}(\text{movies} \times \text{actormovies}).$$

This last rule may be surprising, but these kind of rules occur due to the fact that we are working in a partial (snapshot) database. Furthermore, these kinds of rules could indicate incompleteness of the database wich is also valuable knowledge in the context of data cleaning.

Example 3.19. *We can find 'frequent' genres in which actors play. The rule*

$$\begin{aligned}&\pi_{\text{am.aid}}(\text{actormovies}) \Rightarrow\\ &\quad \pi_{\text{am.aid}}\sigma_{\text{am.mid=gm.mid}\wedge\text{gm.gid=g.gid}\wedge\text{g.gid='3'}}(\text{actormovies} \times \text{genremovies} \times \text{genres})\end{aligned}$$

has 40% confidence, so 40% of the actors play in a 'Documentary' (genre id 3) while the same rule for 'Drama' has 50% confidence.

But also other types of interesting patterns can be discovered. For instance, the following rule has 82% confidence.

$$\begin{aligned}&\pi_{\text{am.aid,am.mid}}\sigma_{\text{am.mid=gm.mid}\wedge\text{gm.gid=g.gid}\wedge\text{g.gid='16'}}(\text{am} \times \text{gm} \times \text{g}) \Rightarrow\\ &\qquad\qquad \pi_{\text{am.aid,am.mid}}\sigma_{\text{am.mid=gm.mid}\wedge\text{gm.gid=g.gid}\wedge\text{g.gid='16'}}(\text{am} \times \text{gm} \times \text{g}).\end{aligned}$$

Intuitively it could indicate that 82% of the actors in genre 'Music' (genre id 16) only play in one movie. But this rule could just as well indicate that one actor plays in 18% of the movies. Examining constant values for actor id (under a low confidence threshold) could allow us to find patterns like the latter.

3.3.2 Quiz Database

The quiz database consists of two relations *quizzes (q)* and *scores (s)*, containing the data about the quiz (who made it, the category,...) and data about the participants (name, result,...) respectively.

Similarly to the IMDB snapshot database we are able to find functional dependencies.

Example 3.20. *We find that the rule*

$$\pi_{\text{q.qid,q.author}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{q.qid}}(\text{quizzes} \times \text{scores})$$

has 100% confidence and represents a functional dependency. The rule

$$\pi_{\text{s.player}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{s.player}}\sigma_{\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores})$$

however, only has 99% confidence and thus again it represents an ***approximate inclusion dependency****. It means that at least one player played a quiz that is not present in the* quizzes *table. Rules like this could indicate that there might be errors in the database, and are therefore very valuable in practice.*

We also discovered other types of patterns of which some examples are given below.

Example 3.21. *To our surprise, the following rule has only 86% confidence:*

$$\pi_{\text{q.qid}}\sigma_{\text{q.author=s.player}\wedge\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{q.qid}}\sigma_{\text{q.author=s.player}\wedge\text{q.qid=s.qid}\wedge\text{s.score='9'}}(\text{quizzes} \times \text{scores})$$

This rule expresses that only in 86% of the cases where a maker plays his own quiz he gets the maximum score of 9.

Example 3.22. *We discovered a rule about a particular player, Benny in this case. We can see that he only has the maximum score in 70% of the quizzes he made himself.*

$$\pi_{\text{q.qid}}\sigma_{\text{q.author=s.player}\wedge\text{q.author='Benny'}\wedge\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{q.qid}}\sigma_{\text{q.author=s.player}\wedge\text{q.author='Benny'}\wedge\text{q.qid=s.qid}\wedge\text{s.score='9'}}(\text{quizzes} \times \text{scores}).$$

For this same player we also discover the following rule having 76% confidence:

$$\pi_{\text{q.qid}}\sigma_{\text{q.category='music'}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{q.qid}}\sigma_{\text{q.category='music'}\wedge\text{q.author='Benny'}\wedge\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores})$$

This rule tells us that of all the quizzes in the category music, Benny has played 76% of them.

Next, we discovered the following rule having 49% confidence.

$$\pi_{\text{s.player}}\sigma_{\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{s.player}}\sigma_{\text{q.author='Benny'}\wedge\text{q.qid=s.qid}}(\text{quizzes} \times \text{scores})$$

This rule tells us that 49% of all players have played a quiz made by Benny.

We also found that Benny has some real fans as the following rule has 98% confidence.

$$\pi_{\text{s.player}}\sigma_{\text{q.qid=s.qid}\wedge\text{q.author='Benny'}}(\text{quizzes} \times \text{scores}) \Rightarrow \pi_{\text{s.player}}\sigma_{\text{q.qid=s.qid}\wedge\text{q.author='Benny'}\wedge\text{s.player='Raf'}}(\text{quizzes} \times \text{scores})$$

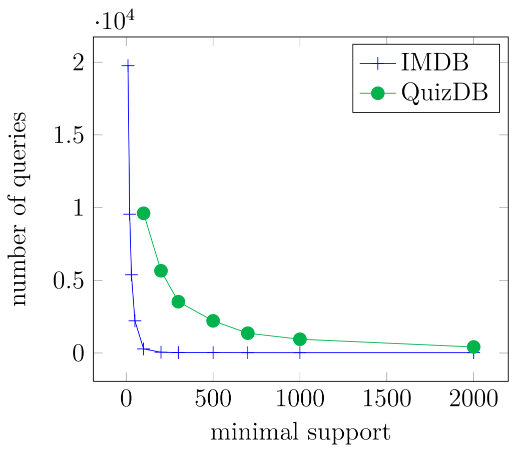
It tells us that the player Raf has played 98% of Benny's quizzes.

3.3.3 Performance

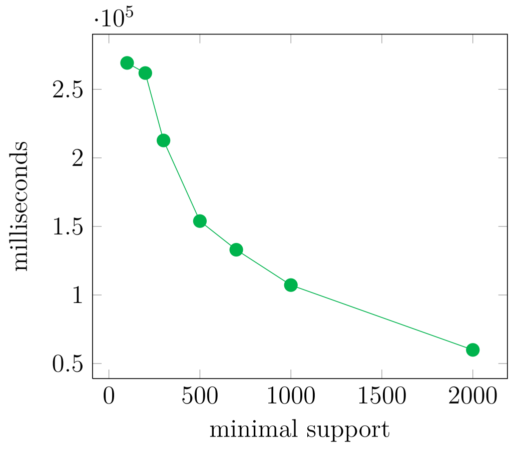
Using our implemented prototype, we conducted some performance measurements using the IMDB snapshot and Quiz databases. The results can be seen in Figure 3.4. Looking at Figure 3.4a we can see that for a lower minimal support the number of patterns generated increases drastically. This is partially due to the fact that an exponential number of combinations of constant values come in to play at these lower support thresholds. The IMDB snapshot database has a higher number of unique constant values compared to the Quiz database which results in quicker exponential behaviour as can be seen in Figure 3.4a. In Figure 3.4b we can also see that for QuizDB the time scales with respect to the number of patterns. For the IMDB dataset in Figure 3.4c this trend is less obvious, since in this case the computation of the join determines the largest part of the timing. Essentially, we conclude that these initial experiments show the feasibility of our approach.

3.4 Dependencies

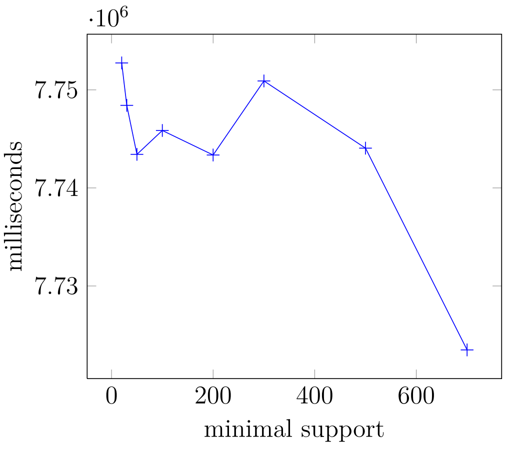
So far we defined the Conqueror algorithm and have shown it discovers interesting association rules over the simple, but appealing subclass of conjunctive queries, called *simple conjunctive queries*. One challenge that remains to be solved is the huge number of generated patterns, especially for lower support values. Part of the volume is inherently due to the relational setting. Mining patterns over multiple relations and several combinations of relations inherently results in more patterns than mining on a single relation. A substantial part of the large number of patterns, however, is due to redundancies induced by dependencies embedded in the data. There is a close relationship between dependencies and redundancies in databases [Ullman, 1988]. In general, a dependency is a constraint on the database, stating that only a subset of all possible instances are valid, *i.e.*, only certain instances reflect a possible state of the real world. This fact implies that there will be some sort of redundancy in legal instances. Knowing some information of the values in the instance and the dependency enables us to deduce other information about the values in the instance. In order not to have these redundancies in a database, one typically decomposes it, based on the dependencies, such that single relations no longer contain redundancies. Knowing the set of dependencies on a database allows us to reduce the number of queries we can generate since many of them will become equivalent under these dependencies. Unfortunately not all real-world databases are necessarily neatly decomposed, and as such will contain redundant information. Moreover, even for some nicely constructed database we might not know the dependencies that hold on it, and therefore cannot exploit them when querying. Luckily, as shown by the examples in Section 3.3, our algorithm is



(a) number of patterns



(b) Quiz database: runtime



(c) IMDB: runtime

Figure 3.4: Experiments for Conqueror with increasing minimal support

capable of detecting previously unknown dependencies. In this section we look at the different kinds of dependencies more closely, as well as at the kind of redundant queries they produce, and more importantly, how we can avoid this. Furthermore, we investigate how we can detect these dependencies and see that we can even use these newly detected dependencies to instantly reduce redundant query generation.

Next to their role in query redundancy, [Boulicaut, 1998] also describes how dependencies can be used to aid in the understanding of data semantics of real-life relational databases, helping experts to analyse what properties actually hold in the data and support the comparison with desired properties. In this section we also consider this point of view and examine dependency types that cannot be used for redundancy removal but are potentially able to provide new semantical insights.

3.4.1 Functional Dependencies

Functional dependencies are probably the most important (and also most known) database dependencies. They are defined as follows [Ullman, 1988]:

Definition 3.7. *Let $R(A_1, \ldots, A_n)$ be a relation scheme, and let X and Y be subsets of $\{A_1, \ldots, A_n\}$. We say $X \to Y$, read "X functionally determines Y" or "Y* ***functionally depends*** *on X" if whatever relation r is the current instance for R, it is not possible that r has two tuples that agree in the components for all attributes in the set X yet disagree in one or more attributes in the set Y.*

One example of a well known use of functional dependency is the definition of a key of a relation. The key of a relation is the minimal attribute set that functionally determines all attributes of that relation. In our setting functional dependencies can induce many redundant queries.

Example 3.23. *Given the scheme $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$, $sch(R_2) = \{C, D, E\}$ and $sch(R_3) = \{F\}$ if we suppose for example the functional dependency $C \to DE$[2] holds, the following queries are essentially equivalent:*

$$\begin{array}{l} \pi_C(R_1 \times R_2 \times R_3) \\ \pi_{CD}(R_1 \times R_2 \times R_3) \\ \pi_{CE}(R_1 \times R_2 \times R_3) \\ \pi_{CDE}(R_1 \times R_2 \times R_3) \end{array}$$

Furthermore, any query that add restrictions with respect to $\pi_C(R_1 \times R_2 \times R_3)$ (e.g., a join or a constant equality) also appears with these equivalent projections. Hence this functional dependency causes many equivalent queries

If we know these functional dependencies, we are able to avoid generating such redundant queries. [Jen et al., 2008] studied the problem of mining all frequent projection-selection queries from a single relation, and assumed that the relation to be mined satisfies a set of functional dependencies. These dependencies were then exploited to remove all redundant queries with respect to these dependencies, from the search space of all queries. To accomplish this, a pre-ordering was defined over queries, and showed to be anti-monotonic towards the support measure. This pre-ordering forms the basis for the definition of equivalence classes that are used for efficient generation of frequent queries, since two equivalent queries are shown to have the same support.

[2]Note, that as is common for functional dependencies, we use concatenation as a notation for union, thus $A_1 \cdots A_n$ is used to represent the set of attributes $\{A_1, \ldots, A_n\}$

Our simple conjunctive queries are more general than the selection-projection queries considered by Jen et al. as we allow for arbitrary joins between multiple relations. In order to successfully avoid the generation of duplicates we therefore generalise the theory to be able to apply the same principles to arbitrary joins of relational tables. This allows us to mine non-redundant simple conjunctive queries, given a collection of functional dependencies over the relations of an arbitrary relational database. In Section 3.5 we generalise our query comparison to take into account functional dependencies, and in Section 3.6 we update our algorithm Conqueror to generate non-redundant queries.

Additionally, we also uncovered, that mining simple conjunctive queries using diagonal containment allows us to detect previously unknown functional dependencies (see Example 3.20 in Section 3.3). This is due to the following strong relationship between support and functional dependency, described in the following proposition.

Proposition 3.3. *Let R be a relation over the attribute set $sch(R)$ and let X and X' be subsets of $sch(R)$. R satisfies $X \rightarrow X'$ if and only if $support(\pi_{XX'}R) = support(\pi_X R)$.*

Proof. If R satisfies $X \rightarrow X'$ then the tuples of R agreeing on X must also agree on X'. This means that the number of tuples with unique values for X is the same as the number of tuples with unique values for XX', and hence $support(\pi_{XX'}R) = support(\pi_X R)$ follows. If on the other hand we know that $support(\pi_{XX'}R) = support(\pi_X R)$, then it cannot be that two tuples that agree on X, do not agree on X'. If this was the case then $support(\pi_{XX'}R) > support(\pi_X R)$. Thus it follows that $X \rightarrow X'$ holds in R. $\square$

Note that we cannot actually state, as required by the definition, that this dependency holds for all instances, since we are only investigating one. The functional (and other) dependencies we are able to discover are therefore always instance dependent.

We are not only capable of detecting functional dependencies on relations, but also functional dependencies that hold on arbitrary joins of relations.

Example 3.24. *Considering the schema $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, for $Q = \pi_{AD}\sigma_{A=C}(R_1 \times R_2)$ and $Q' = \pi_A\sigma_{A=C}(R_1 \times R_2)$, considering an instance $\mathcal{I}$ of $\mathcal{D}$ for which $support(Q) = support(Q')$ indicates that $\sigma_{A=C}(R_1 \times R_2)(\mathcal{I})$ satisfies the functional dependency $A \rightarrow D$.*

Based on Proposition 3.3 we introduce a novel technique to discover previously unknown functional dependencies during the mining process, and *immediately* exploit them in reducing the number of redundant frequent queries in the output.

As stated, we do not only consider functional dependencies over the original relations of the database, but also over *any join* of multiple relations. Such functional dependencies are again used to limit the final collection of queries to only those that are non-redundant with respect to both the given and discovered collections of functional dependencies. In Section 3.6 we describe the changes we made to the basic Conqueror algorithm in order to enable this new functionality.

Our main purpose for the discovery of functional dependencies is their pruning potential. In contrast, the discovery of functional dependencies as a goal itself has been extensively studied the past years and several algorithms have been developed for this specific purpose. They can be roughly classified according to the approach they take [Yao & Hamilton, 2008]: the candidate generate-and-test approach [Bell & Brockhausen, 1995, Huhtala et al., 1999, Novelli & Cicchetti, 2001], the minimal cover approach [Flach & Savnik, 1999, Lopes et al., 2000, Wyss et al., 2001], and the formal concept analysis approach [Baixeries, 2004, Lopes et al., 2002]. Although the Conqueror algorithm could also be used to discover functional dependencies as a goal, our algorithm tackles the more general query mining problem. These algorithms are specialised, and therefore also much more efficient for the specific task of discovering functional dependencies.

3.4.2 Conditional Functional Dependencies

Recent studies have shown that dirty data creates huge costs both due to the usage of faulty knowledge as in man-hours spent cleaning up data manually. It is with this argument that [Bohannon et al., 2007] introduce the notion of conditional functional dependencies.

Definition 3.8. *A **conditional functional dependency** φ is of the form $(R : X \to Y, T_p)$ here R is the relation where the dependency holds, X and Y are sets of attributes from R, $X \to Y$ is a standard dependency* embedded in *φ, and T_p is a* pattern tableau*, where for each attribute A in X or Y and each tuple $t_p \in T_p, t[A]$ is either a constant 'a' from the domain of A or an unnamed variable '_'.*

A data tuple t is said to match a pattern tuple t_p if for all attributes A of the tuple $t[A] = t_p[A]$ or $t_p[A] =$ '_'.

A relation R satisfies the conditional functional dependency φ if for each pair of tuples t_1, t_2 in R and for each tuple t_p in the pattern tableau T_p of φ it holds that if $t_1[X] = t_2[X]$ and both match $t_p[X]$, then $t_1[Y] = t_2[Y]$ and both must match $t_p[Y]$.

Next to proving the effectiveness of conditional functional dependencies in detecting and repairing inconsistencies of data [Fan et al., 2008, Fan et al., 2009] introduce several methods for the discovery of conditional functional dependencies. The CFDMiner algorithm is based on closed itemset mining techniques and only mines constant conditional functional dependencies (no unnamed variable allowed). CTANE and FastCFD are algorithms developed to mine general conditional functional dependencies, based on the well known regular functional dependency mining algorithms TANE [Huhtala et al., 1999] and FastFD [Wyss et al., 2001]. [Chiang & Miller, 2008] present another algorithm for conditional functional dependency discovery, and additionally, search for approximate conditional functional dependencies to find dirty data (We cover approximate dependencies in Section 3.4.3).

Since simple conjunctive queries are capable of expressing patterns with attribute values, and we already proved we are able to detect functional dependencies by comparing support of queries, finding conditional functional dependencies is also possible using our approach. Indeed, the detection of conditional functional dependencies can be done in much the same way as functional dependencies, as the following proposition shows.

Proposition 3.4. *Given a pattern tuple t_p, let the selection condition $\sigma_{X=t_p[X]}$ denote the conjunction of $A = a$ for each constant 'a' in t_p. The conditional functional dependency $(R : X \to Y, T_p)$ holds if and only if for all t_p in T_p*

$$\mathit{support}(\pi_{XY}\sigma_{X=t_p[X]}(R)) = \mathit{support}(\pi_X\sigma_{X=t_p[X]}(R))$$

and

$$\mathit{support}(\pi_{sch(R)}\sigma_{X=t_p[X]}(R)) = \mathit{support}(\pi_{sch(R)}\sigma_{X=t_p[X]\wedge Y=t_p[Y]}(R)).$$

Proof. The conditional functional dependency $(R : X \to Y, T_p)$ holds if for each t_p in T_p and for each pair of tuples t_1, t_2 if $t_1[X] = t_2[X]$ and both match $t_p[X]$, then $t_1[Y] = t_2[Y]$. This holds if and only if the number of unique values for X matching $t_p[X]$ is equal to the number of unique values of XY where X matches $t_p[X]$, or put in other words if $\mathit{support}(\pi_{XY}\sigma_{X=t_p[X]}(R)) = \mathit{support}(\pi_X\sigma_{X=t_p[X]}(R))$. Furthermore, it must additionally hold for each such pair, that both $t_1[Y]$ and $t_2[Y]$ must match $t_p[Y]$. This holds if and only if the number of tuples where X matches $t_p[X]$ equals the number of tuples where both $t_1[Y]$ and $t_2[Y]$ must match $t_p[Y]$. This is the same as $\mathit{support}(\pi_{sch(R)}\sigma_{X=t_p[X]}(R)) = \mathit{support}(\pi_{sch(R)}\sigma_{X=t_p[X]\wedge Y=t_p[Y]}(R))$. □

In order to include this detection in our algorithm we need to add a conditional functional dependency discovery check. We search for rules of the type $\pi_X\sigma_{X=t_p[X]}(R) \Rightarrow \pi_{XY}\sigma_{X=t_p[X]}(R)$ with 100% confidence (antecedent and confident have equal support). These rules represent a special kind of conditional functional dependency which we refer to as *dependency-only* conditional functional dependencies. In essence, they represent the conditional functional dependencies where the Y part of the pattern tuple only contains an unnamed variable (*i.e.*, a wildcard). Intuitively these are 'regular' functional dependencies that hold, but only for the specified X values. Alongside these we can also discover 100% association rules of the type $\pi_{sch(R)}\sigma_{X=t_p[X]}(R) \Rightarrow \pi_{sch(R)}\sigma_{X=t_p[X]\wedge Y=t_p[Y]}(R)$. We call these *condition-only* conditional functional dependencies. Here we are in fact stating a trivial dependency $sch(R) \rightarrow sch(R)$, but we are requiring, via the pattern tuple, that when $X = t_p[X]$, it must hold that $Y = t_p[Y]$. Both these detections can be easily added to the rule generation procedure or as postprocessing. Using Proposition 3.4, both types of conditional functional dependencies can then be combined to form general conditional functional dependencies as presented in Definition 3.8.

Example 3.25. *Considering the schema $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, if we find that*

$$support(\pi_{CD}\sigma_{C=`c'}(R_1 \times R_2)) = support(\pi_{CDE}\sigma_{C=`c'}(R_1 \times R_2))$$

we have found the conditional functional dependency ($R_2 : CD \rightarrow E$,(c, _ || _))[3]. *Furthermore, if we additionally find that*

$$support(\pi_{CDE}\sigma_{C=`c'}(R_1 \times R_2)) = support(\pi_{CDE}\sigma_{(C=`c')\wedge(E=`e')}(R_1 \times R_2))$$

than we can conclude that the conditional functional dependency ($R_2 : CD \rightarrow E$,(c, _ || e)) must hold.

The definition of *support* for a conditional functional dependency [Fan et al., 2009], is the number of tuples in the relation, that match the pattern. In our approach this is the same as $support(\pi_R\sigma_{X=t_p[X]\wedge Y=t_p[Y]}(R))$.

It has been clearly shown that conditional functional dependencies are useful in practice, and it is therefore nice that our conjunctive query mining technique is able to discover them. Despite this nice result, our algorithm has a more general goal, and for that reason it can never be as efficient as the mentioned specialised approaches. Moreover, these methods also mine minimal sets of conditional functional dependencies, while this is not true in our case. We must note, however, that analogous to the discovery of functional dependencies we are also capable of detecting conditional functional dependencies over arbitrary joins of relations, a feature not present in these specialised approaches. Furthermore, similarly to [Chiang & Miller, 2008] we can also have a notion of approximate conditional functional dependencies. We touch on this subject in Section 3.4.3.

Theoretically we could also apply conditional functional dependencies to prune redundant queries, since for any more specific query the dependency holds. Since, however, conditional functional dependencies include constant values, and this part of the algorithm is already executed efficiently using SQL in the database, we chose not to include any pruning based on conditional functional dependencies in our current algorithm (Section 3.6).

[3] We use the notation of [Fan et al., 2009], where left hand side and right hand side attributes of the functional dependency are separated with ‘ || ’ in the pattern tuple.

3.4.3 Approximate Functional Dependencies

An approximate functional dependency is a functional dependency that almost holds. For example, a persons gender is approximately determined by their first name. Approximate functional dependencies arise in many real world databases where there is a natural dependency between some attributes, but due to errors or exceptions in some tuples, the dependency does not hold for the whole relation. Similar to conditional functional dependencies (which also represent approximate functional dependencies since they also hold for part of the relation), approximate functional dependencies have many applications in data cleaning, as they can potentially expose errors or missing information in the database. Also from a pure semantical standpoint, approximate functional dependencies can result in valuable insight. [Huhtala et al., 1999] give the motivating example of a database relation of chemical compounds. Here a approximate functional dependency relating various structural attributes to carcinogenicity could provide valuable hints to biochemists for potential causes of cancer, although evidently an expert analysis still needs to be performed.

The discovery of approximate functional dependencies has received considerable interest, and various techniques for the discovery of approximate functional dependencies exist [Huhtala et al., 1999, Lopes et al., 2002, Matos & Grasser, 2004, Engle & Robertson, 2008, Sánchez et al., 2008]. Most of them make use of the g_3 measure [Kivinen & Mannila, 1995], defined as follows:

$$g_3(X \to Y, R) = \frac{|R| - max\{|S| \mid S \subseteq R, S \vDash X \to Y\}}{|R|}$$

Intuitively it represents the minimal fraction of tuples in the instance of a relation R, that violate the dependency $X \to Y$. We know from Proposition 3.3 that the dependency $X \to Y$ holds if $\mathit{support}(\pi_{XY}(R)) = \mathit{support}(\pi_X(R))$. If, however, $X \to Y$ does not hold, the difference $\mathit{support}(\pi_{XY}(R)) - \mathit{support}(\pi_X(R))$ should give some measure of how many tuples violate the dependency. Our support measure eliminates duplicates, therefore this difference represents the number of unique values for XY in R for which the dependency does not hold. This number is always lower than the minimal number of tuples for which the dependency does not hold. Therefore, using our support measure we can obtain a lower bound to the g_3 measure:

$$g_3(X \to Y, R) \geq \frac{\mathit{support}(\pi_{XY}(R)) - \mathit{support}(\pi_X(R))}{|R|}$$

This lower bound, which we refer to as g_3^s, gives the number of unique XY values that do not satisfy the dependency $X \to Y$ relative to all tuples in R. In general we can conclude that if the value for g_3^s is high we know $X \to Y$ is no approximate functional dependency. A low value of g_3^s, however, does not guarantee that it is. Finding all dependencies with a value for g_3^s below a certain error threshold will only give us the set of *possible* approximate functional dependencies.

Depending on the amount of duplication of XY and X values, the g_3^s is more or less close to the g_3 value. On that matter we must note that duplication of tuples satisfying the dependency does not affect g_3^s since they cancel each other out in the difference of support values. Only duplicates in the violating XY values affect the measure, since they are counted only once in g_3^s. The relative support $\mathit{support}(\pi_{XY}(R))/|R|$ gives a measure for the total amount of duplication of XY values. The closer it is to 1, the smaller the amount of duplication of XY in R. Hence, to take into account duplication we divide g_3^s by $\mathit{support}(\pi_{XY}(R))/|R|$ which results in the following measure:

$$\begin{aligned} g_3^c(X \to Y, R) &= \frac{\mathit{support}(\pi_{XY}(R)) - \mathit{support}(\pi_X(R))}{\mathit{support}(\pi_{XY}(R))} \\ &= 1 - \frac{\mathit{support}(\pi_X(R))}{\mathit{support}(\pi_{XY}(R))} \\ &= 1 - \mathit{confidence}(\pi_{XY}(R) \to \pi_X(R)) \end{aligned}$$

This measure gives us the fraction of unique values for XY in R that violate the dependency $X \to Y$. Unfortunately, this measure can now be higher than g_3, making it no longer a lower bound. The g_3 measure is not the only measure used to express approximateness of functional dependencies, and many other measures, such as *InD*, have been considered [Giannella & Robertson, 2004, Sánchez et al., 2008]. The g_3^c measure can be seen as a new error measure for approximate functional dependencies, that is closely related to g_3 but invariant with respect to duplication of tuples. We will refer to dependencies with a g_3^c below a certain maximal error threshold as *confident functional dependencies.* In fact, formulated this way, they are no more than a certain kind of conjunctive query association rules, and detecting them comes down to identifying the association rules $Q_1 \Rightarrow Q_2$ where $\bowtie(Q_1) = \bowtie(Q_2) = \emptyset$, $\sigma(Q_1) = \sigma(Q_2) = \emptyset$ and $\pi(Q_1) = (\pi(Q_2) \cup X)$. Our algorithm therefore generates such confident functional dependencies, and could potentially be optimised for this specific case. However, as is the case with (conditional) functional dependencies, many specialised algorithms exists [Huhtala et al., 1999, Matos & Grasser, 2004, Engle & Robertson, 2008] which are better suited for the purpose. Although they do not provide support for our g_3^c measure, they could potentially be adapted to do so.

We can also consider *approximate* conditional functional dependencies. If we change the constraint of the rules above to allow for $\sigma(Q_1) = \sigma(Q_2) \neq \emptyset$, we can find approximate dependency-only conditional functional dependencies. Similarly we can also consider approximate condition-only conditional functional dependencies. Unfortunately, since these dependencies do not hold 100%, they can no longer be combined to form general approximate conditional functional dependencies, since we do not know if the dependency-only and the condition-only dependencies hold for the same set of tuples.

Intuitively, approximate functional dependencies can also entail a lot of redundancies.

Example 3.26. *If we suppose $C \to DE$ holds with 98% confidence in the setting of Example 3.23, then the queries*

$$\begin{array}{l}\pi_C(R_1 \times R_2 \times R_3)\\ \pi_{CD}(R_1 \times R_2 \times R_3)\\ \pi_{CE}(R_1 \times R_2 \times R_3)\\ \pi_{CDE}(R_1 \times R_2 \times R_3)\end{array}$$

will all have approximately the same support.

However, since we are not capable of deriving these support values from the approximate confidence, we cannot prune these queries. Therefore, we cannot use approximate functional dependencies to reduce the amount of queries generated if we want lossless pruning. Approximate functional dependencies could be used if only approximate results are required. However, this is not part of our goal, and we therefore do not consider this setting any further.

3.4.4 Inclusion Dependencies

Next to functional dependencies, there is another often considered type of dependencies, namely the inclusion dependencies. Inclusion dependencies state that the values for certain set attributes in one relation, must all occur as values for a set of attributes in another relation. Formally we can define them as follows [Abiteboul et al., 1995].

Definition 3.9. *Given a scheme $\mathcal{D}$, where R and S are relations in $\mathcal{D}$, and $X \subseteq sch(R)$ and $Y \subseteq sch(S)$, then the instance $\mathcal{I}$ satisfies the* ***inclusion dependency*** *$R[X] \subseteq S[Y]$ if $\pi_X(R)(\mathcal{I}) \subseteq \pi_Y(S)(\mathcal{I})$.*

Although it has received fewer attention than the discovery of functional dependencies, some algorithms have been also developed for the discovery of inclusion dependencies [Kantola et al., 1992, Marchi et al., 2002, Marchi & Petit, 2003, Marchi et al., 2004, Koeller & Rundensteiner, 2003, Koeller & Rundensteiner, 2006]. In the simplest case these algorithms are based on the levelwise approach, but for larger patterns, optimisations are proposed. The Conqueror algorithm also allows for the detection of inclusion dependency by means of query comparison (see Example 3.18 in Section 3.3). This is stated as follows:

Proposition 3.5. *Let R and S be relations respectively defined over $sch(R)$ and $sch(S)$, such that $sch(R) \cap sch(S) = \emptyset$. If X and Y are subsets of $sch(R)$ and $sch(S)$, respectively, then the inclusion dependency $R[X] \subseteq S[Y]$ is satisfied if and only if $support(\pi_X(R \times S)) = support(\pi_X \sigma_{X=Y}(R \times S))$.*

Proof. If R and S satisy $R[X] \subseteq S[Y]$ then all values for X are included in $\pi_Y(S)$, this implies that the the distinct results for $\pi_X(R \times S)$ equal the distinct results for $\pi_X \sigma_{X=Y}(R \times S)$, and hence $support(\pi_X(R \times S)) = support(\pi_X \sigma_{X=Y}(R \times S))$ follows trivially.

If $support(\pi_X(R \times S)) = support(\pi_X \sigma_{X=Y}(R \times S))$ holds then for each tuple t_R in R it must hold that there exists a tuple in t_S in S such that $t_R[X] = t_S[Y]$, hence $\pi_X(R) \subseteq \pi_Y(S)$ and $R[X] \subseteq S[Y]$ follows. □

Next to the intrinsic interestingness of these kinds of patterns, inclusion dependencies can also result in redundant queries. Unfortunately [Casanova et al., 1984] showed that, although there are simple complete axiomatizations for functional dependencies alone and for inclusion dependencies alone, there is no complete axiomatization for functional and inclusion dependencies taken together. Additionally, they showed that the inference problem for inclusion dependencies alone is PSPACE-complete, in contrast with the polynomial time complexity for the functional dependency inference problem. So although inclusion dependencies do imply additional redundant queries, the equivalence of such queries cannot be computed efficiently [Johnson & Klug, 1984]. For this reason, we do not generalise query comparison to include inclusion dependencies, choosing instead only to include functional dependencies, as detailed in Section 3.5. Luckily, in the specific case of foreign-keys, inclusion dependencies can be used to avoid some query evaluations. This optimisation is discussed in Section 3.4.5.

We can also consider the notion of approximate inclusion dependencies. For this purpose the measure g'_3, has been introduced [Marchi et al., 2002]. As the name implies, it is based on the g_3 measure. Intuitively it corresponds to the proportion of distinct values of X one has to remove from R to obtain a database $\mathbf{d}'$ such that $\mathbf{d}' \models R[X] \subseteq S[Y]$.

$$g'_3(R[X] \subseteq S[Y], \mathbf{d}) = \\ 1 - \frac{max\{|\pi_X(R')| \mid R' \subseteq R, (\mathbf{d} - \{R\}) \cup \{R'\} \models R[X] \subseteq S[Y]\}}{|\pi_X(R)|}$$

Since this measure makes use of distinct values, we can also formulate this measure using simple conjunctive queries and our support measure as follows:

$$\begin{aligned} g'_3(R[X] \subseteq S[Y], \mathbf{d}) &= \frac{\mathit{support}(\pi_X(R \times S)) - \mathit{support}(\pi_X \sigma_{X=Y}(R \times S))}{\mathit{support}(\pi_X(R))} \\ &= 1 - \frac{\mathit{support}(\pi_X \sigma_{X=Y}(R \times S))}{\mathit{support}(\pi_X(R \times S))} \\ &= 1 - \mathit{confidence}(\pi_X(R \times S) \Rightarrow \pi_X \sigma_{X=Y}(R \times S)) \end{aligned}$$

Formulated this way, we can see that similar to approximate functional dependencies, approximate inclusion dependencies are no more than a certain kind of conjunctive query association rules where the error threshold ϵ corresponds to an $(1 - \epsilon)$ minimal confidence threshold. Also similar to functional dependencies, our algorithm could be modified to only generate such rules, but will hardly be as efficient as the mentioned specialised (approximate) inclusion dependency mining algorithms. However, our algorithm has the benefit that it puts the discovery of both (approximate) functional and (approximate) inclusion dependencies together in a broader context of conjunctive query association rules.

3.4.5 Foreign-keys

In relational databases, a *foreign-key* is an attribute (or attribute combination) in one relation R whose values are required to match those of the primary key of some relation S [Date, 1986]. Typically foreign-keys are defined using referential integrity constraints provided in the database [Codd, 1970]. However, as stated before, such information is not necessarily present in a given real-world database. Luckily, such a foreign-key constraint is essentially made up of two dependencies: a key dependency and an inclusion dependency. Since Conqueror also discovers inclusion dependencies in combination with functional dependencies, it is able to discover foreign-keys.

Proposition 3.6. *Let R and S be relations respectively defined over $sch(R)$ and $sch(S)$, such that $sch(R) \cap sch(S) = \emptyset$. If X and Y are subsets of $sch(R)$ and $sch(S)$, respectively, then Y is a foreign-key referencing the key X if and only if*

$$\mathit{support}(\pi_{sch(R)}(R \times S)) = \mathit{support}(\pi_X(R \times S)) = \mathit{support}(\pi_X \sigma_{X=Y}(R \times S))$$

Proof. Following the definition, Y is a foreign-key referencing X if and only if the functional dependency $X \rightarrow sch(R)$ and the inclusion dependency $S[Y] \subseteq R[X]$ hold. The proposition then follows from Proposition 3.3 and 3.5. □

This knowledge of foreign-keys can now be used to avoid unneeded query evaluation.

Example 3.27. *Considering the schema $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, assume that the functional dependency $A \to B$ holds, and that we are given the inclusion dependency $R_1[C] \subseteq R_2[A]$. If all frequent queries involving respectively R_1 and R_2 are known, then consider the following query $\pi_{ABCDE}\sigma_{A=C}(R_1 \times R_2)$. This is clearly a join performed along the key A of R_1 and the foreign-key C in R_2. Because of the functional dependency and $A = C$ we know*

$$support(\pi_{ABCDE}\sigma_{A=C}(R_1 \times R_2)) = support(\pi_{CDE}\sigma_{A=C}(R_1 \times R_2))$$

Knowing that $R_1[C] \subseteq R_2[A]$ we can also conclude that

$$support(\pi_{CDE}\sigma_{A=C}(R_1 \times R_2)) = support(\pi_{CDE}(R_1 \times R_2))$$

Therefore, if $\pi_{CDE}(R_1 \times R_2)$ is frequent, $\pi_{ABCDE}\sigma_{A=C}(R_1 \times R_2)$ is known to be frequent without *having to compute its support. Moreover, we have a similar situation for some of the projections of $Q = \pi_{ABCDE}\sigma_{A=C}(R_1 \times R_2)$. More precisely, it can be seen that $support(\pi_{ABCD}Q) = support(\pi_{CD}(R_1 \times R_2)$, $support(\pi_{ABCE}Q) = support(\pi_{CE}(R_1 \times R_2))$ and $support(\pi_{ABC}Q) = support(\pi_{C}(R_1 \times R_2))$. Consequently, the supports of $\pi_{ABCD}Q$, $\pi_{ABCE}Q$ and $\pi_{ABC}Q$ should not be computed, as well as the supports of some of selections of these queries. This results in a significant reduction of the number of query evaluations, that is not possible by only considering $A \to B$. Notice, however, that for example, $support(\pi_{BE}Q)$ must be computed as BE is not involved in the key-foreign-key constraint. In fact, given the projection generation tree shown in Figure 3.5, only for the **bold** projection queries, the support is known.*

We discuss how we use this knowledge in avoiding query evaluation in Section 3.6, where we describe the modifications to the Conqueror algorithm.

3.4.6 Conclusion

To conclude, dependencies result in the generation of many redundant queries. We therefore update our algorithm in the next sections in order to make use of these dependencies and generate non-redundant queries. Furthermore, we showed that using comparisons of conjunctive queries, we are able to detect previously unknown functional and key dependencies. In the upcoming sections we explain how we do this algorithmically as well as how we maximally exploit the newly discovered dependencies in order to avoid generating or evaluating queries that are redundant with respect to these dependencies.

Additionally, we showed that we are also capable of discovering conditional and approximate dependencies and showed how our confidence measure relates to the

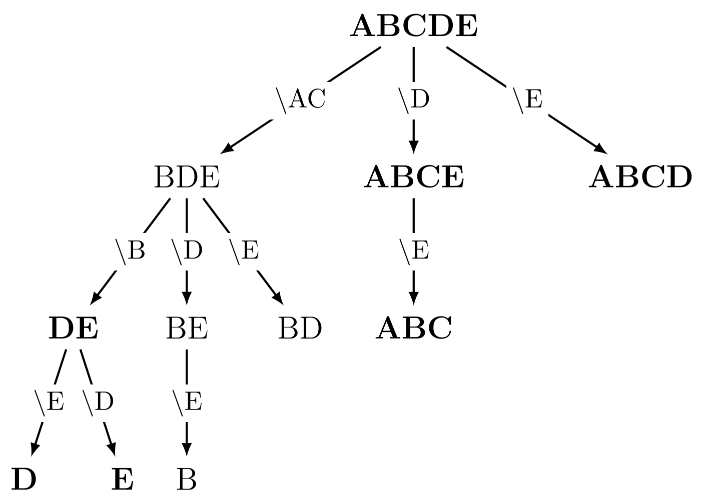


Figure 3.5: Projection generation tree for Example 3.27

existing measures used for this task in the field. Although our general algorithm is not as efficient as specialised conditional and approximate functional and inclusion dependency mining algorithms that have been developed, it does discover all of them together in a broader context of conjunctive query association rules, using measures that closely relate to those already used in the field.

3.5 Generalising Query Comparison

We now update our query comparison to take into account functional dependencies. First let us introduce the following definition of a join query, to simplify further notation:

Definition 3.10. *We call Q a* ***join query*** *if $\sigma(Q)$ is the empty condition and if $\pi(Q)$ is the set of all attributes of all relation names occurring in $\bowtie(Q)$. Given a simple conjunctive query Q, we denote by $J(Q)$ the join query such that $\bowtie(J(Q)) = \bowtie(Q)$.*

Assume that we are given functional dependencies over $\mathcal{D}$. More precisely, each R_i is associated with a set of functional dependencies over $sch(R_i)$, denoted by $\mathcal{FD}_i$, and the set of all functional dependencies defined in $\mathcal{D}$ is denoted by $\mathcal{FD}$, we now define the pre-ordering $\preceq$ to compare queries, which generalises both diagonal containment as well as the pre-order of [Jen et al., 2008]:

Definition 3.11. *Let $Q_1 = \pi_{X_1}\sigma_{F_1}(R_1 \times \cdots \times R_n)$ and $Q_2 = \pi_{X_2}\sigma_{F_2}(R_1 \times \cdots \times R_n)$ be two simple conjunctive queries. Denoting by Y_i the schema of Q_i^σ, for $i = 1, 2$, $Q_1 \preceq Q_2$ holds if*

1. *$\bowtie(Q_1) \subseteq \bowtie(Q_2)$*
2. *$J(Q_2)(\mathcal{I})$ satisfies $X_1Y_2 \rightarrow X_2$ and $Y_2 \rightarrow Y_1$, and*
3. *the tuple $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$.*

Example 3.28. *Considering the schema $\mathcal{D}$ consisting of $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, assume that $\mathcal{FD}_1 = \emptyset$ and $\mathcal{FD}_2 = \{C \rightarrow D, E \rightarrow D\}$, and let $Q_1 = \pi_{AD}\sigma_{(A=C)\wedge(E=e)}R$ and $Q_2 = \pi_C\sigma_{(A=C)\wedge(D=d)}R$. We have $\bowtie(Q_1) = \bowtie(Q_2)$ and $J(Q_1) = J(Q_2) = \pi_{ABCDE}\sigma_{(A=C)}R$. Then, if $\mathcal{I}$ is an instance of $\mathcal{D}$, $J(Q_1)(\mathcal{I})$ satisfies $\mathcal{FD}$. Moreover, due to the equality defining $\bowtie(Q_1)$, $J(Q_1)(\mathcal{I})$ also satisfies the two functional dependencies $A \rightarrow C$ and $C \rightarrow A$. Therefore, $J(Q_1)(\mathcal{I})$ satisfies $CE \rightarrow AD$ and $E \rightarrow D$, and so, if $(d, e) \in \pi_{DE}J(Q_1)(\mathcal{I})$, by Definition 3.11, $Q_2 \preceq Q_1$.*

This relation is a pre-ordering as we now prove.

Proposition 3.7. *The relation $\preceq$ is a pre-ordering over the simple conjunctive queries on $\mathcal{D}$*

Proof. It is easy to see that $\preceq$ is reflexive, therefore we only show the transitivity of $\preceq$. Let $Q_1 = \pi_{X_1}\sigma_{F_1}R$ and $Q_2 = \pi_{X_2}\sigma_{F_2}R$ and $Q_3 = \pi_{X_3}\sigma_{F_3}R$ be such that $Q_1 \preceq Q_2 \preceq Q_3$. We now need to prove that $Q_1 \preceq Q_3$. First of all, since the subset relation is transitive, we know that $\bowtie(Q_1) \subseteq \bowtie(Q_3)$. First we prove that $J(Q_3)(\mathcal{I})$ satisfies $X_1Y_3 \rightarrow X_3$ and $Y_3 \rightarrow Y_1$. Since $\bowtie(Q_2) \subseteq \bowtie(Q_3)$, $J(Q_3)(\mathcal{I})$ satisfies all functional dependencies that hold in $J(Q_2)(\mathcal{I})$. Therefore, it is easy to see that $J(Q_3)(\mathcal{I})$ satisfies $Y_3 \rightarrow Y_1$, since we know $Y_3 \rightarrow Y_2$ and $Y_2 \rightarrow Y_1$. Furthermore, using transitivity we find $X_1Y_3 \rightarrow X_1Y_3Y_2$ since we know $Y_3 \rightarrow Y_2$. Then we know $X_1Y_2Y_3 \rightarrow X_1Y_2Y_3X_2$ since we know $X_1Y_2 \rightarrow X_2$. Finally we know $X_1Y_2Y_3X_2 \rightarrow X_3$ since we know$X_2Y_3 \rightarrow X_3$, and thus using decomposition we can conclude $X_1Y_3 \rightarrow X_3$. Second we prove that the tuple $Q_1^\sigma Q_3^\sigma$ is in $\pi_{Y_1Y_3}J(Q_3)(\mathcal{I})$. Since we know $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$ and that $\bowtie(Q_2) \subseteq \bowtie(Q_3)$ and $\bowtie(Q_1) \subseteq \bowtie(Q_3)$, we also know that $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_3)(\mathcal{I})$. Using the fact that $Q_2^\sigma Q_3^\sigma$ is in $\pi_{Y_2Y_3}J(Q_3)(\mathcal{I})$ we can then conclude that $Q_1^\sigma Q_3^\sigma$ is in $\pi_{Y_1Y_3}J(Q_3)(\mathcal{I})$. □

We note that, although the above pre-ordering is instance dependent, this has no impact regarding computational issues, as our algorithm only considers tuples that occur in the relations from which frequent queries are mined. Also remark that, in contrast to diagonal containment, this pre-ordering is *not* a partial order. Suppose for example that in the context of Example 3.1 A, B is the key of R_1, then trivially the FDs $A \to B$ and $B \to A$ hold. Therefore both $\pi_A(R_1 \times R_2) \preceq \pi_B(R_1 \times R_2)$ and $\pi_B(R_1 \times R_2) \preceq \pi_A(R_1 \times R_2)$ hold, but it is clear that $\pi_A(R_1 \times R_2) \neq \pi_B(R_1 \times R_2)$.

Similar to diagonal containment and the pre-ordering from [Jen et al., 2008], we can prove that the pre-ordering satisfies the important basic property that the support of queries is anti-monotonic with respect to it, that is, for all simple conjunctive queries Q_1 and Q_2 such that $Q_1 \preceq Q_2$, we have $\mathit{support}(Q_2) \leq \mathit{support}(Q_1)$.

Lemma 3.2. *If $X \to Y$ is a functional dependency that holds for $Q(\mathcal{I})$.*

1. *$\mathit{support}(\pi_Y Q) \leq \mathit{support}(\pi_X Q)$*
2. *$\mathit{support}(\sigma_{X=x} Q) \leq \mathit{support}(\sigma_{Y=y} Q)$, where y is the tuple over Y associated with x through $X \to Y$ in $Q(\mathcal{I})$*

Proof.

1. Since $X \to Y$ holds for Q there exists a total, onto function from $\pi_X Q(\mathcal{I})$ to $\pi_Y Q(\mathcal{I})$, thus $\mathit{support}(\pi_Y Q) \leq \mathit{support}(\pi_X Q)$
2. For every tuple t in $\sigma_{X=x} Q(\mathcal{I})$, $t.X = x$. Thus, $t.Y = y$, showing that t is in $\sigma_{Y=y} Q(\mathcal{I})$. Therefore we have $\mathit{support}(\sigma_{X=x} Q) \leq \mathit{support}(\sigma_{Y=y} Q)$.

$\square$

Lemma 3.3. *If $\bowtie_1 \subseteq \bowtie_2$ then $\mathit{support}(\pi_X \sigma_{F \wedge \bowtie_2} R) \leq \mathit{support}(\pi_X \sigma_{F \wedge \bowtie_1} R)$*

Proof. Since $\bowtie_1$ is a set of conditions of the type $A = A'$, and $\bowtie_1 \subseteq \bowtie_2$, we know that $\bowtie_2$ contains at least all the conditions of $\bowtie_2$. Therefore every tuple t in the answer of $\pi_X \sigma_{F \wedge \bowtie_2} R$ will satisfy all the conditions of $\pi_X \sigma_{F \wedge \bowtie_1} R$ and therefore $\pi_X \sigma_{F \wedge \bowtie_2} R(\mathcal{I}) \subseteq \pi_X \sigma_{F \wedge \bowtie_1} R$, from which we can conclude that $\mathit{support}(\pi_X \sigma_{F \wedge \bowtie_2} R) \leq \mathit{support}(\pi_X \sigma_{F \wedge \bowtie_1} R)$. $\square$

Proposition 3.8. ***(Support Anti-Montonicity)*** *For all simple conjunctive queries on $\mathcal{D}$, if $Q_1 \preceq Q_2$ then $\mathit{support}(Q_2) \leq \mathit{support}(Q_1)$*

Proof. If Q_2^σ is the empty tuple, it follows from $Y_2 \to Y_1$ that Y_1 and Y_2 are empty. Now $X_1 \to X_2$ holds, then from Lemma 3.2 it follows that $\mathit{support}(\pi_{X_2} Q_1) \leq \mathit{support}(Q_1)$ using Lemma 3.3 we know that $\mathit{support}(Q_2) \leq \mathit{support}(\pi_{X_2} Q_1)$, transitively we conclude that $\mathit{support}(Q_2) \leq \mathit{support}(Q_1)$.

If Q_2^σ is not the empty tuple, $\sigma_{\sigma(Q_2) \wedge \bowtie(Q_2)}(\mathcal{I})$ satisfies $X_1 \to Y_2$, and as $X_1 Y_2 \to X_2$, $\sigma_{\sigma(Q_2) \wedge \bowtie(Q_2)}(\mathcal{I})$ satisfies $X_1 \to X_2$. Using Lemma 3.2, point 1 we obtain

$$\mathit{support}(\pi_{X_2} \sigma_{\sigma(Q_2) \wedge \bowtie(Q_2)} R) \leq \mathit{support}(\pi_{X_1} \sigma_{\sigma(Q_2) \wedge \bowtie(Q_2)} R).$$

As we know $Y_2 \to Y_1$ and $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$, using Lemma 3.2 point 2, we obtain

$$\mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_2)\wedge\bowtie(Q_2)}R) \leq \mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_1)\wedge\bowtie(Q_2)}R).$$

Thus transitively we obtain

$$\mathit{support}(\pi_{X_2}\sigma_{\sigma(Q_2)\wedge\bowtie(Q_2)}R) \leq \mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_1)\wedge\bowtie(Q_2)}R).$$

Finally using Lemma 3.3 we can say that

$$\mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_1)\wedge\bowtie(Q_2)}R) \leq \mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_1)\wedge\bowtie(Q_1)}R).$$

Thus transitively we conclude

$$\mathit{support}(\pi_{X_2}\sigma_{\sigma(Q_2)\wedge\bowtie(Q_2)}R) \leq \mathit{support}(\pi_{X_1}\sigma_{\sigma(Q_1)\wedge\bowtie(Q_1)}R)$$

or $\mathit{support}(Q_2) \leq \mathit{support}(Q_1)$. □

Of course, this anti-monotonicity is used in our algorithms to prune infrequent queries, in much the same way as infrequent queries were pruned using diagonal containment (see Section 3.1.1). Similarly, the pre-ordering $\preceq$ induces an equivalence relation, denoted by $\sim$ and defined as follows: given two simple conjunctive queries Q_1 and Q_2, $Q_1 \sim Q_2$ holds if $Q_1 \preceq Q_2$ and $Q_2 \preceq Q_1$.

As a consequence of anti-monotonicity of the support, if $Q_1 \sim Q_2$ holds then $\mathit{support}(Q_1) = \mathit{support}(Q_2)$. Therefore, only *one* computation per equivalence class modulo $\sim$ is enough to determine the support of *all* queries in that class.

Proposition 3.9. *Let $Q_1 = \pi_{X_1}\sigma_{F_1}R$ and $Q_2 = \pi_{X_2}\sigma_{F_2}R$ be two simple conjunctive queries, $Q_1 \sim Q_2$ holds if and only if $\bowtie(Q_1) = \bowtie(Q_2)$ and $(X_1Y_1)^+ = (X_2Y_2)^+$, $Y_1^+ = Y_2^+$ and $Q_1^\sigma Q_2^\sigma \in \pi_{Y_1Y_2}J(Q_1)(\mathcal{I})$.*

Proof. For $Q_1 \sim Q_2$ to hold, it must be that $\bowtie(Q_1) \subseteq \bowtie(Q_2)$ and $\bowtie(Q_2) \subseteq \bowtie(Q_1)$, thus $\bowtie(Q_1) = \bowtie(Q_2)$. Assuming that $J(Q_1)(\mathcal{I})$ satisfies the functional dependencies of a given set FD, since $\bowtie(Q_1) = \bowtie(Q_2)$, $J(Q_2)(\mathcal{I})$ also satisfies this set. Since it must hold that $Y_2 \to Y_1$ and $Y_1 \to Y_2$, $Y_1^+ = Y_2^+$ follows. Similarly since it must hold that $X_1Y_2 \to X_2$ and $X_2Y_1 \to X_1$, together with $Y_1^+ = Y_2^+$, it follows that $(X_1Y_1)^+ = (X_2Y_2)^+$. □

Using this proposition, we can define the representative of the equivalence class of a query Q as the query Q^+, such that $\pi(Q^+) = ((\pi(Q) \cup \mathit{sch}(Q^\sigma))^+ \setminus \mathit{sch}(Q^\sigma)^+)^+$, $\bowtie(Q^+) = \bowtie(Q)$ and $\sigma(Q^+)$ is the selection condition corresponding to the super tuple of Q^σ, denoted by $(Q^\sigma)^+$, defined over $\mathit{sch}(Q^\sigma)^+$, and that belongs to $\pi_{\mathit{sch}(Q^\sigma)^+}J(Q)(\mathcal{I})$.

We notice that for such a query Q^+ it can never be that $\pi(Q) \subseteq \mathit{sch}(Q^\sigma)$. However, it is easy to see that the support of Q is 1, which is meant to be less than the minimum support threshold. Therefore, the queries Q of interest are such that

$$\pi(Q) = ((\pi(Q) \cup sch(Q^\sigma))^+ \setminus sch(Q^\sigma))^+ \text{ and}$$
$$sch(Q^\sigma) = sch(Q^\sigma)^+.$$

In what follows, such queries are said to be *closed queries* and the closed query equivalent to a given query Q is denoted by Q^+. Moreover, we denote by $\mathcal{Q}^+$ the set of all closed queries.

Example 3.29. *Referring back to the queries Q_1 and Q_2 of Example 3.28, it is easy to see that they do* not *satisfy the restrictions above. It can be seen that these queries are not closed, and thus, they are not considered in our algorithm. But as $J(Q_1)(\mathcal{I})$ satisfies $C \to D$, $E \to D$, $A \to C$ and $C \to A$, the closed queries Q_1^+ and Q_2^+ defined below are processed instead.*

$$Q_1^+ = \pi_{ACD}\sigma_{(A=C)\wedge(E=e)}R$$
$$Q_2^+ = \pi_{ACD}\sigma_{(A=C)\wedge(E=e)\wedge(D=d)}R.$$

Considering only such queries in our updated algorithm (Section 3.6), reduces the size of the output set of frequent queries. This size is reduced even further, since dependencies other than those specified in $\mathcal{FD}$ are discovered during the processing of the algorithms. This point is addressed in more details in Section 3.6.2. Furthermore, similar to diagonal containment (Proposition 3.3), our new comparison operator forms a partial order over closed queries. This allows us to generate queries in a similar way as the Conqueror algorithm, as we explain in Section 3.6.

3.5.1 Non-Redundant Association Rules

We originally defined association rules based on diagonal containment, which was also the basis for query comparison and generation in Conqueror. Now, we have defined a new query comparison relation taking functional dependencies into account, and as such we also redefine association rules. Using the pre-ordering $\preceq$, we define the strict partial ordering $\prec$ on $\mathcal{Q}^+$ as $Q_1^+ \prec Q_2^+$ if $Q_1^+ \preceq Q_2^+$ and not $Q_1^+ \sim Q_2^+$.

Definition 3.12. *An* ***association rule*** *is of the form $Q_1^+ \Rightarrow Q_2^+$, such that Q_1^+ and Q_2^+ are both closed simple conjunctive queries and $Q_1^+ \prec Q_2^+$.*

Since we are generating equivalence classes, similar to diagonal containment, we only consider association rules based upon closed queries. Such rules are, unfortunately, not always diagonal rules.

Example 3.30. *Supposing the dependency $C \to E$ holds, the following closed queries are contained according to the pre-order*

$$\pi_{AB}(R_1 \times R_2) \prec \pi_{ABCE}\sigma_{A=C}(R_1 \times R_2)$$

however

$$\pi_{ABCE}\sigma_{A=C}(R_1 \times R_2) \not\subseteq^{\Delta} \pi_{AB}(R_1 \times R_2)$$

because $\{A, B, C, E\} \not\subseteq \{A, B\}$*. However, there are equivalent non-closed queries in the equivalence classes of the considered queries, such that the diagonal containment does hold:*

$$\pi_{AB}\sigma_{A=C}(R_1 \times R_2) \subseteq^{\Delta} \pi_{AB}(R_1 \times R_2).$$

However, this is not true for all queries, still considering the above setting, it holds that

$$\pi_{AB}(R_1 \times R_2) \prec \pi_{E}\sigma_{A=C}(R_1 \times R_2)$$

and obviously

$$\pi_{E}\sigma_{A=C}(R_1 \times R_2) \not\subseteq^{\Delta} \pi_{AB}(R_1 \times R_2).$$

Unfortunately there is no equivalent query such that it does hold. However, we notice that

$$\pi_{AB}(R_1 \times R_2) \prec \pi_{ABCE}\sigma_{A=C}(R_1 \times R_2) \prec \pi_{E}\sigma_{A=C}(R_1 \times R_2).$$

In this case, it is possible to find equivalent queries

$$\pi_{AB}\sigma_{A=C}(R_1 \times R_2) \subseteq^{\Delta} \pi_{AB}(R_1 \times R_2)$$

$$\pi_{E}\sigma_{A=C}(R_1 \times R_2) \subseteq^{\Delta} \pi_{ABCE}\sigma_{A=C}(R_1 \times R_2).$$

We write $Q_1^+ \prec^1 Q_2^+$ if there does not exist a Q^+ such that $Q_1^+ \prec Q^+ \prec Q_2^+$. So, similar to diagonal containment, we only consider *basic rules*, that is $Q_1^+ \Rightarrow Q_2^+$ if $Q_1^+ \prec^1 Q_2^+$. We now show that in that case, we are able to find an equivalent diagonally contained rule.

Proposition 3.10. $Q_1^+ \prec^1 Q_2^+$ *if and only if one of the following holds the following holds:*

1. $\bowtie(Q_1^+) = \bowtie(Q_2^+)$ *and* $(Q_1^+)^\sigma = (Q_2^+)^\sigma$ *and* $\pi(Q_2^+) \subset \pi(Q_1^+)$*, and there does not exist a schema* X *such that* $X^+ = X$ *and* $\pi(Q_2^+) \subset X \subset \pi(Q_1^+)$*.*
2. $\bowtie(Q_1^+) = \bowtie(Q_2^+)$ *and* $(Q_1^+)^\sigma$ *is a strict subtuple of* $(Q_2^+)^\sigma$ *and* $\pi(Q_2^+) = ((\pi(Q_1^+) \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2^+)^\sigma))^+$ *and* $\nexists X \subset \pi(Q_1^+) : \pi(Q_2^+) = ((X \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2^+)^\sigma))^+$*, and there does not exist a tuple* y *over* Y *such that* $Y^+ = Y$*, and* $(Q_1^+)^\sigma$ *is a strict subtuple of* y *and* y *is a strict subtuple of* $(Q_2^+)^\sigma$*.*
3. $\bowtie(Q_1^+) \subset \bowtie(Q_2^+)$ *and* $(Q_2^+)^\sigma = ((Q_1^+)^\sigma)^+$ *and* $\pi(Q_2^+) = (\pi(Q_1^+))^+$ *and* $\nexists X \subset \pi(Q_1^+) : X^+ = \pi(Q_2^+)$*, and there does not exist a* Q *such that* $\bowtie(Q_1^+) \subset \bowtie(Q) \subset \bowtie(Q_2^+)$

Proof. In all of these cases it is easy to see that if the requirements are met, $Q_1^+ \prec^1 Q_2^+$ follows. Now we prove the other direction. We know $Q_1^+ \prec^1 Q_2^+$. Then it must hold that either (1) $\bowtie(Q_1^+) = \bowtie(Q_2^+)$ or (2) $\bowtie(Q_1^+) \subset^1 \bowtie(Q_2^+)$ since otherwise a query Q^+, such that $Q_1^+ \prec Q^+ \prec Q_2^+$, would exist.

In the following, we denote $\pi(Q_i^+)$ as X_i and $sch((Q_i^+)^\sigma)$ as Y_i. For the case (1) where $\bowtie(Q_1^+) = \bowtie(Q_2^+)$ it must hold that $X_1Y_2 \rightarrow X_2$ and $Y_2 \rightarrow Y_1$ and it cannot be that there is a Q with the same join, such that $Y_2 \rightarrow Y \rightarrow Y_1$ and $X_1Y \rightarrow X$ and $XY_2 \rightarrow X_2$. From this it follows that either $Y_1 = Y_2$ or $Y_1 \subset^1 Y_2$ since these are both closed. Since Definition 3.11 states that $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$, it follows that it must be that either (1.1) $(Q_1^+)^\sigma = (Q_2^+)^\sigma$ holds or that (1.2) $(Q_1^+)^\sigma$ is a strict subtuple of $(Q_2^+)^\sigma$. We now prove that these only possible cases (1.1), (1.2) and (2) match the described cases of the proposition:

1. If $Q_1^+ \prec^1 Q_2^+$ and $(Q_1^+)^\sigma = (Q_2^+)^\sigma$ and $\bowtie(Q_1^+) = \bowtie(Q_2^+)$, then it must hold that $X_1Y_2 \rightarrow X_2$. According to the definition $X_1 = ((X_1Y_1)^+ \setminus Y_1)^+$, since from $(Q_1^+)^\sigma = (Q_2^+)^\sigma$ it follows that $Y_1 = Y_2$, and thus that $X_1 = ((X_1Y_2)^+ \setminus Y_2)^+$. From $Q_1^+ \prec^1 Q_2^+$ follows that $X_1Y_2 \rightarrow X_2$ and thus $X_1 = ((X_1Y_2X_2)^+ \setminus Y_2)^+$. Since $X_2 = ((X_2Y_2)^+ \setminus Y_2)^+$, it is clear that $X_2 \subseteq X_1$, *i.e.*, $\pi(Q_2^+) \subseteq \pi(Q_1^+)$.

   Furthermore, $\pi(Q_2^+) \neq \pi(Q_1^+)$ since otherwise $Q_1 = Q_2$ and then $Q_1^+ \not\prec Q_2^+$, thus $\pi(Q_2^+) \subset \pi(Q_1^+)$. Also there cannot exist an X such that $\pi(Q_2^+) \subset X \subset \pi(Q_1^+)$, because then $Q_1 \prec \pi_X Q_1 \prec Q_2$ and then $Q_1^+ \not\prec Q_2^+$.

2. According to the definition of closed query $X_2 = ((X_2Y_2)^+ \setminus Y_2)^+$. If $Q_1^+ \prec^1 Q_2^+$ and $(Q_1^+)^\sigma$ is a strict subtuple of $(Q_2^+)^\sigma$ and $\bowtie(Q_1^+) = \bowtie(Q_2^+)$, it must hold that $X_1Y_2 \rightarrow X_2$, it also holds that $X_1Y_2 \rightarrow X_2Y_2$, thus considering $X_2' = ((X_1Y_2)^+ \setminus Y_2)^+$, it must hold that $X_2 \subseteq X_2'$. However, it cannot be that $X_2 \subset X_2'$ since otherwise there would exist a Q where $\pi(Q) = X_2'$ and $\sigma(Q) = \sigma(Q_2)$ and $\bowtie(Q) = \bowtie(Q_2^+)$, where $Q_1 \prec Q \prec Q_2$. Thus $X_2 = X_2'$. Moreover, it must hold that $\nexists X \subset X_1 : X_2 = ((X \cup Y_2)^+ \setminus Y_2)^+$, otherwise there would exist a $Q = \pi_X Q_1$, since $Q_1 \prec \pi_X Q_1 \prec Q_2$. Thus it must hold that $\pi(Q_2^+) = ((\pi(Q_1^+) \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2^+)^\sigma))^+$ and $\nexists X \subset \pi(Q_1^+) : \pi(Q_2^+) = ((X \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2^+)^\sigma))^+$.

   Furthermore, it cannot be that there exists a tuple y over Y such that $Y^+ = Y$, and $(Q_1^+)^\sigma$ is a strict subtuple of y and y is a strict subtuple of $(Q_2^+)^\sigma$. Otherwise $Q_1 \prec \sigma_{Y=y} Q_1 \prec Q_2$ and $Q_1^+ \not\prec Q_2^+$.

3. If $Q_1^+ \prec^1 Q_2^+$ it follows that from the definition where it states that $Y_2 \rightarrow Y_1$ must hold in $J(Q_2)$, that $Y_1^+ \subseteq Y_2$ under the functional dependencies of $J(Q_2)$. If, however, $Y_1^+ \subset Y_2$ then, since we know $\bowtie(Q_1^+) \subset \bowtie(Q_2^+)$ it follows

that $Q_1 \prec \sigma_{\sigma(Q_1)^+} Q_2 \prec Q_2$, which is a contradiction. Thus it must hold that $Y_2 = Y_1^+$ or stated otherwise that $(Q_2^+)^\sigma = ((Q_1^+)^\sigma)^+$.

Similar to the previous case it follows that $X_2 = ((X_1 Y_2)^+ \setminus Y_2)^+$. Since we also know that $Y_2 = Y_1^+$, it also follows that $X_2 = ((X_1 Y_1)^+ \setminus Y_1)^+$. Since this is closed under $\Join(Q_2^+)$ it follows that $X_2 = X_1^+$. Moreover it must hold that $\nexists X \subset X_1$ such that $X^+ = X_2$, otherwise there would exist $Q = \pi_X Q_1$, since $Q_1 \prec \pi_X Q_1 \prec Q_2$. Thus it must hold that $\pi(Q_2^+) = (\pi(Q_1^+))^+$ and $\nexists X \subset \pi(Q_1^+) : X^+ = \pi(Q_2^+)$.

Furthermore, there cannot exist a Q such that $\Join(Q_1^+) \subset \Join(Q) \subset \Join(Q_2^+)$, otherwise $Q_1 \prec \sigma_{\Join(Q)} Q_1 \prec Q_2$ and $Q_1^+ \not\prec Q_2^+$

□

Proposition 3.11. *For all closed queries Q_1^+, Q_2^+ it holds that if $Q_1^+ \prec^1 Q_2^+$ then $\pi_{\pi(Q_1^+)\cap\pi(Q_2^+)} Q_2^+ \subseteq^\Delta Q_1^+$ and $\pi_{\pi(Q_1^+)\cap\pi(Q_2^+)} Q_2^+ \sim Q_2^+$*

Proof. $\pi(Q_1^+) \cap \pi(Q_2^+) \subseteq \pi(Q_2^+)$ trivially holds, so we have to prove $Q_2' \subseteq Q_1^+$ where $Q_2' = \pi_{\pi(Q_1^+)\cap\pi(Q_2^+)} Q_2^+$. Since Q_1^+ and Q_2^+ are closed $sch((Q_i^+)^\sigma) = sch((Q_i^+)^\sigma)^+$ holds, and we know $sch((Q_2^+)^\sigma) \rightarrow sch((Q_1^+)^\sigma)$, thus it must follow that $sch((Q_1^+)^\sigma) \subseteq sch((Q_2^+)^\sigma)$. Together with the fact that the tuple $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1 Y_2} J(Q_2)(\mathcal{I})$, it follows that $\sigma(Q_1^+) \subseteq \sigma(Q_2^+)$. Because we also know $\Join(Q_1) \subseteq \Join(Q_2)$, it follows that $F_1 \subseteq F_2$. Thus all tuples in the answer of Q_2' will satisfy the conditions of Q_1^+, and hence $Q_2' \subseteq Q_1^+$ follows.

We now prove $\pi_{\pi(Q_1^+)\cap\pi(Q_2^+)} Q_2^+ \sim Q_2^+$. $Q_2^+ \preceq \pi_{\pi(Q_1^+)\cap\pi(Q_2^+)} Q_2^+$ is trivial since $\pi(Q_2^+) sch((Q_2^+)^\sigma) \rightarrow \pi(Q_1^+) \cap \pi(Q_2^+)$. We now have to prove that $(\pi(Q_1^+) \cap \pi(Q_2^+) sch((Q_2^+)^\sigma) \rightarrow \pi(Q_2^+)$. Following Proposition 3.10, if $Q_1^+ \prec^1 Q_2^+$ we have three cases. In the first case we have $\pi(Q_2^+) \subset \pi(Q_1^+)$, then $\pi(Q_1^+) \cap \pi(Q_2^+) = \pi(Q_2^+)$ so $(\pi(Q_1^+) \cap \pi(Q_2^+) sch((Q_2^+)^\sigma) \rightarrow \pi(Q_2^+)$ follows trivially. In the third case we have $\pi(Q_2^+) = (\pi(Q_1^+))^+$, thus $\pi(Q_1^+) \subseteq \pi(Q_2^+)$, and $\pi(Q_1^+) \cap \pi(Q_2^+) = \pi(Q_1^+)$ and thus using $\pi(Q_1^+) sch((Q_2^+)^\sigma) \rightarrow \pi(Q_2^+)$, again it is proven. In the second case we have $\pi(Q_2) = (\pi(Q_1^+) \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2)^\sigma)^+$. Thus

$$\begin{array}{rl} & (\pi(Q_1^+) \cap \pi(Q_2^+)) \cup sch((Q_2)^\sigma \\ = & (\pi(Q_1^+) \cap (\pi(Q_1^+) \cup sch(Q_2^\sigma))^+ \setminus sch((Q_2)^\sigma)^+ \cup sch((Q_2)^\sigma \\ = & ((\pi(Q_1) \cup sch((Q_2)^\sigma)^+ \setminus sch((Q_2)^\sigma))^+ \cup sch((Q_2)^\sigma) \end{array}$$

From this we can conclude that $\pi(Q_1) \cup sch((Q_2)^\sigma) \subseteq (\pi(Q_1^+) \cap \pi(Q_2^+)) \cup sch((Q_2)^\sigma$ and thus $(\pi(Q_1) \cap \pi(Q_2)) \cup sch((Q_2)^\sigma) \rightarrow \pi(Q_1) \cup sch((Q_2)^\sigma$. Since $\pi(Q_1^+) \cup sch((Q_2)^\sigma \rightarrow \pi(Q_2^+)$, it also holds in this case.

□

Proposition 3.12. *For all canonical simple conjunctive queries Q_1, Q_2 it holds that if $Q_1 \subseteq^{\Delta} Q_2$ then $Q_2 \preceq Q_1$*

Proof. If $Q_1 \subseteq^{\Delta} Q_2$ then $\pi(Q_1) \subset \pi(Q_2)$ thus it holds that $\pi(Q_2) \rightarrow \pi(Q_1)$ and thus trivially that $\pi(Q_2)sch(Q_1^{\sigma}) \rightarrow \pi(Q_1)$. Since the queries are canonical, according to Lemma 3.1 it follows that $\sigma(Q_1) \subseteq \sigma(Q_2)$. Then it follows that $sch(Q_2) \rightarrow sch(Q_2)$ and that $Q_1^{\sigma}Q_2^{\sigma}$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$. Thus $Q_2 \preceq Q_1$. □

Proposition 3.13. *Let $Q_1^+ \Rightarrow Q_2^+$ be a confident association rule. Then there exist n confident basic rules of the form $Q_i^+ \Rightarrow Q_{i+1}^+$ $(i = 1, \ldots, n)$ such that $Q_{n+1}^+ = Q_2^+$.*

Proof. If $Q_1^+ \Rightarrow Q_2^+$ is not a basic rule, it is possible to find basic rules as specified in the proposition. Moreover, confidence($Q_1^+ \Rightarrow Q_2^+$) is the product of confidence($Q_i^+ \Rightarrow Q_{i+1}^+$) for $i = 1, \ldots, n$, and so, confidence($Q_1^+ \Rightarrow Q_2^+$) $\leq$ confidence($Q_i^+ \Rightarrow Q_{i+1}^+$). As we assume $Q_1^+ \Rightarrow Q_2^+$ to be confident, then so are the basic rules ($Q_i^+ \Rightarrow Q_{i+1}^+$) for $i = 1, \ldots, n$. Thus, the proof is complete. □

Proposition 3.14. *If $Q_1 \Rightarrow Q_2$ is a basic association rule over canonical simple conjunctive queries then there exist n basic association rules over closed simple conjunctive queries of the form $Q_i^+ \Rightarrow Q_{i+1}^+$ $(i = 1, \ldots, n)$ such that $Q_{n+1}^+ = Q_2^+$, and Q_1^+ and Q_2^+ are equivalent to Q_1 and Q_2 respectively.*

If $Q_1^+ \Rightarrow Q_2^+$ is a basic association rule over closed simple conjunctive queries, then there exists an association rule $Q_1 \Rightarrow Q_2$ over canonical simple conjunctive queries such that Q_1^+ and Q_2^+ are equivalent to Q_1 and Q_2 respectively.

Proof. The proof follows from Propositions 3.11, 3.12 and 3.13. □

This essentially proves that our basic rules form a cover over all diagonally contained rules. We must note, however, that our definition of association rules implies that functional dependencies cannot be output as association rules, since functional dependencies are 'embedded' in the frequent queries computed in our approach. Therefore we consider the found functional dependencies and association rules together as the complete (association) rule output of the algorithm.

3.6 Algorithm: Conqueror$^+$

In this section, we present an updated algorithm, which we call Conqueror$^+$ (given as Algorithm 3.7), for mining frequent queries under functional and foreign-key dependencies. Conqueror$^+$ follows the basic three-loop structure of Conqueror but with the following additions in each loop:

- **Join loop:** Generate all instantiations of F, without constants, *i.e.* $\bowtie(Q)$, in a breadth-first manner, using restricted growth to represent partitions. *Every such partition gives rise to a join query JQ and functional dependencies of its ancestors are inherited.*

- **Projection loop:** For each generated partition, all projections of the corresponding join query JQ, *i.e* $\pi(Q)$, are generated in a breadth-first manner, and their frequency is tested against the given instance $\mathcal{I}$. *During this loop, functional dependencies and foreign-keys are discovered and used to prune the search space.*

- **Selection loop:** For each frequent projection-join query, constant assignments, *i.e.* $\sigma(Q)$, are added to F in a breadth-first manner. *Morever, here again, functional dependencies are used to prune the search space.*

3.6.1 Handling Functional Dependencies

First of all a (possibly empty) set of functional dependencies $\mathcal{FD}$ can be specified as input. This set is used for the first queries processed by our algorithm (*i.e.,* the relations of the database instance) on line 3 of Algorithm 3.7.

Join Loop

A new addition to the *join loop* is the inheritance of functional dependencies shown on lines 13-14. In Conqueror$^+$ a given join query JQ is associated with a set of functional dependencies, denoted by $\mathcal{FD}_{JQ}$, and built up in Algorithm 3.7 as follows.

First, when $\bowtie$(Q) is the restricted growth string 1, every instantiated relation $R_i(\mathcal{I})$ in the database is pushed in *Queue* (lines 2 and 6, Algorithm 3.8), associated with the set of functional dependencies $\mathcal{FD}_i$ (see line 3, Algorithm 3.7).

Then, at the next level, the considered restricted growth strings represent join conditions of the form $(R_i.A = R_j.A')$. Let JQ be the corresponding join query. If $R_i = R_j$ then, $JQ(\mathcal{I})$ satisfies the functional dependencies of $\mathcal{FD}_i$, since JQ is a selection of R_i. Moreover, $JQ(\mathcal{I})$ also satisfies $R_i.A \rightarrow R_i.A'$ and $R_i.A' \rightarrow R_i.A$, due to the selection condition. Thus, $\mathcal{FD}_{JQ}$ is set to $\mathcal{FD}_i \cup \{R_i.A \rightarrow R_i.A', R_i.A' \rightarrow R_i.A\}$. Similarly, if $R_i \neq R_j$, then JQ is a join of R_i and R_j. Thus, $JQ(\mathcal{I})$ clearly satisfies the functional dependencies of $\mathcal{FD}_i \cup \mathcal{FD}_j$, as well as $R_i.A \rightarrow R_j.A'$ and $R_j.A' \rightarrow R_i.A$. In this case, we set $\mathcal{FD}_{JQ} = \mathcal{FD}_i \cup \mathcal{FD}_j \cup \{R_i.A \rightarrow R_j.A', R_j.A' \rightarrow R_i.A\}$. The assignment of $\mathcal{FD}_{JQ}$ in these two cases is processed in lines 13-16 of Algorithm 3.7.

We notice that, at this stage, $\pi(JQ)$ is either the set $sch(R_i)$ (if $R_i = R_j$) or $sch(R_i) \cup sch(R_j)$ (if $R_i \neq R_j$), and thus, $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.

Algorithm 3.7 Conqueror$^+$

Input: Database $\mathcal{D}$, dependencies $\mathcal{FD}$, minmal support *minsup*, most specific join *msj*

Output: Frequent Queries FQ

```
1: $\bowtie$(Q) := "1" //initial restricted growth string
2: for all $R_i$ in $\mathcal{D}$ do
3:   $\mathcal{FD}_{\mathrm{Q}} := \mathcal{FD}_i$
4:   push(Queue, $R_i$)
   //Join Loop
5: while not Queue is empty do
6:   JQ := pop(Queue)
7:   if $\bowtie$(JQ) does not represent a cartesian product then
8:     FQ := FQ $\cup$ ProjectionLoop(JQ)
9:   children := RestrictedGrowth($\bowtie$(JQ), $m$)
10:  for all rgs in children do
11:    if join defined by rgs is not more specific than msj then
12:      $\bowtie$(JQC) := rgs
13:      for all PJQ such that $\bowtie(\mathrm{JQC}) = \bowtie(\mathrm{PJQ}) \wedge (R_i.A = R_j.A')$ do
14:        $\mathcal{FD}_{\mathrm{JQC}} := \mathcal{FD}_{\mathrm{JQC}} \cup \mathcal{FD}_{\mathrm{PJQ}}$
15:        if $\bowtie$(PJQ) = "1" then
16:          $\mathcal{FD}_{\mathrm{JQC}} := \mathcal{FD}_{\mathrm{JQC}} \cup \{R_i.A \to R_j.A', R_j.A' \to R_i.A\}$
17:      blocks(JQC) := blocks(JQ) where the blocks containing $R_i.A$ and $R_j.A'$
         are merged
18:      push(Queue, JQC)
19: return FQ
```

In the general case, when considering the lattice of all join queries, which is in fact the lattice of all partitions of U, at a given level, the join query JQ is generated from join queries PJQ in the previous level by setting $\bowtie(JQ)$ to $\bowtie(PJQ) \wedge (R_i.A = R_j.A')$, and by augmenting $\pi(PJQ)$ accordingly. Therefore, $JQ(\mathcal{I})$ satisfies the dependencies of $\mathcal{FD}_{PJQ}$, and thus, $\mathcal{FD}_{JQ}$ is set to be the union of all $\mathcal{FD}_{PJQ}$ where PJQ allows to generate JQ (see lines 13-14 of Algorithm 3.7). Consequently, assuming that $\pi(PJQ)$ is closed under $\mathcal{FD}_{PJQ}$ clearly entails that $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.

Thus, for every join query JQ, $\pi(JQ)$ is closed under those functional dependencies of $\mathcal{FD}_{JQ}$ that belong to $\mathcal{FD}$ or that are obtained through the connected blocks of *blocks*(JQ). Moreover, as covered next, the discovered functional dependencies in the projection loop of JQ preserve this property, because these new dependencies are defined with attributes in $\pi(JQ)$ only. Therefore, for every join query JQ, $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.

Projection Loop

First of all we generalise the definition of blocks to include mutually dependent attributes. In the setting of Conqueror we defined *blocks*(Q) to be the partition induced by the join condition of Q. In the setting of Conqueror$^+$ we additionally define *blocks*$^+(Q)$ as the partition induced by mutually dependent attributes. An attribute A is **mutually dependent** with an attribute A' if $A \to A'$ and $A' \to A$ hold, and we denote this $A \leftrightarrow A'$.

Example 3.31. *Given the scheme $\mathcal{D}$ consisting of sch$(R_1) = \{A, B\}$, sch$(R_2) = \{C, D, E\}$ and where the functional dependencies $A \to B$, $B \to A$ hold. Now suppose we are considering the join $\bowtie(Q) = R_1.B = R_2.C$. Then blocks$(Q) = \{\{A\}, \{B, C\}, \{D\}, \{E\}\}$ as was shown before. Now since the join $\bowtie(Q) = R_1.B = R_2.C$ holds we actually know $B \to C$ and $C \to B$ must hold for Q, combined with the given dependencies, this results in blocks$^+(Q) = \{\{A, B, C\}, \{D\}, \{E\}\}$*

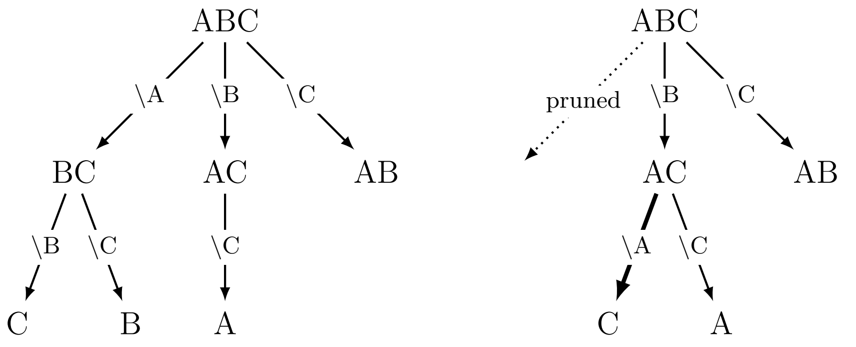
Like in Conqueror, in order to generate candidate projection-join queries in the *projection loop*, we construct the set *torem* as the set of blocks to be removed. In Conqueror$^+$ we are considering blocks of *blocks*$^+(PQ)$, and the removal is achieved in such a way that no block functionally depending on another can be removed, as stated on line 26 of Algorithm 3.8. Constructing the candidate projection-join queries in this way makes sure that, given a join query JQ, only those projection-join queries PQ such that $\pi(PQ)$ is closed under $\mathcal{FD}_{JQ}$ are considered. This is so because the first projection-join query considered in the projection loop is the join query JQ, and we have just shown that for join queries $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$. Then, assuming that $\pi(PQ)$ is closed under $\mathcal{FD}_{JQ}$, when generating the candidate projection-join queries of the next level (lines 19-29, Algorithm 3.8), if $\pi(PQ) \setminus \beta$ is not closed, then the closure of $\pi(PQ) \setminus \beta$ under $\mathcal{FD}_{JQ}$ contains at least one attribute in β, and so all attributes in β (because, for all A and A' in β, $A \to A'$ and $A' \to A$ are in $\mathcal{FD}_{JQ}$). As a consequence of the standard closure algorithm [Ullman, 1988], $\mathcal{FD}_{JQ}$ must contain dependencies $Z \to \beta'$ such that $Z \subseteq \pi(PQ) \setminus \beta$ and $\beta' \cap \beta \neq \emptyset$. Since we enforce this condition on line 26, all frequent projection-join queries PQ mined in our algorithms are such that $\pi(PQ)$ is closed under $\mathcal{FD}_{JQ}$. Since the selection loops processed in Algorithm 3.9 do not change the projection schemas, this also holds for all selection-projection-join queries. We recall from Section 3.5 that considering queries Q such that $\pi(Q)$ is closed under $\mathcal{FD}_{J(Q)}$ (where $J(Q)$ is the join query associated to Q) is an important feature of our approach that avoids the generation and evaluation of redundant queries.

However, since our generation is based on a fixed order of U, when defining the set *torem* we need to take care that we reconsider blocks β for removal when

Algorithm 3.8 Conqueror$^+$ Projection Loop

```
Input: Conjunctive Query Q
1: if $\bowtie$(Q) = “1” then
2:   $\pi$(Q) := $sch(R_i)$ //Q is the query $R_i$
3: else
4:   $\pi(Q)$ := union of blocks$^+$(Q)
5:   ForeignKeyDiscovery(Q)
6: push(Queue, Q)
7: FPQ := $\emptyset$
8: while not Queue is empty do
9:   PQ := pop(Queue)
10:  $\mathcal{KFK}$(Q) := ForeignKeyHandling(PQ)
11:  if not $\mathcal{KFK}$(PQ) then
12:    if monotonicty(PQ) then //monotonicity check
13:      support(PQ) := EvaluateSupport(PQ)
14:    else
15:      support(PQ) := 0
16:  if support(PQ) $\geq$ minsup then
17:    FPQ := FPQ $\cup$ {PQ}
18:    FunctionalDependencyDiscovery(PQ,FPQ)
19:    removed := blocks in blocks$^+$(PQ) not in $\pi$(PQ)
20:    dep := blocks $\beta$ such that there exists $Z \rightarrow \beta$ in $\mathcal{FD}_{\mathrm{Q}}$
21:    deprem := dep $\cap$ removed
22:    if deprem = $\emptyset$ then
23:      torem := dep $\cup$ blocks Y in blocks$^+$(PQ) > last of removed
24:    else
25:      torem := blocks $\beta$ in blocks$^+$(PQ) > last of deprem
26:    torem := blocks $\beta$ of torem such that there is no $Z \rightarrow \beta'$ in $\mathcal{FD}$(Q)
       such that Z $\subseteq \pi$(PQ)$\setminus\beta$ and $\beta' \cap \beta \neq \emptyset$
27:    for all $\beta_i \in$ torem do
28:      $\pi$(PQC) := $\pi$(PQ) with block $\beta_i$ removed //$\pi$(PQC) is closed under $\mathcal{FD}$(Q)
29:      push(Queue, PQC)
30: FQ := FQ $\cup$ FPQ
31: for all PQ $\in$ FPQ do
32:   if PQ is not marked then
33:     FQ := FQ $\cup$ SelectionLoop(PQ)
34: return FQ
```

the dependency $Z \rightarrow \beta$ no longer holds in the case that $Z \nsubseteq \pi(PQ)$ (lines 22-25, Algorithm 3.8). We illustrate this using the following example.



(a) Projection tree of {A,B,C}

(b) Projection tree of {A,B,C} when $B \rightarrow A$ holds

Figure 3.6: Projection generation-trees of {A,B,C}

Example 3.32. *When no dependencies are present, the generation of projections of attributes $\{A, B, C\}$ would result in the projection generation tree shown in Figure 3.6a, where all levels are shown.*

If dependency $B \rightarrow A$ is present, we do not consider A for removal when generating the second level. However, in order to generate the last level, we do need to remove A when condidering AC in order to generate C. This results in the projection generation tree shown in Figure 3.6b.

We must note, that although the current algorithm is capable of dealing with mutual dependencies consisting of single blocks, mutual dependencies of multiple attributes still pose a problem.

Example 3.33. *Suppose we are considering the scheme of Example 3.31, but now the dependencies $ACE \rightarrow B$, $ACE \rightarrow D$, $BD \rightarrow A$, $BD \rightarrow C$ and $BD \rightarrow E$ hold, or also $ACE \leftrightarrow BD$. Starting from the most general projection $\pi(Q) = ABCDE$ one would not be able to generate any sub-projections since every block is dependent on some blocks of $\pi(Q)$. Still CDE (AB removed), ADE (BC removed), ABE (CD removed) and ABC (DE removed) are valid closed sub-projections of $\pi(Q)$ that should be considered.*

So we can conclude that in the presence of mutual dependencies over multiple attributes (blocks), our current algorithm is not complete. In Section 3.10 we briefly discuss our proposed changes to resolve the issue.

Algorithm 3.9 Conqueror$^+$ Selection Loop

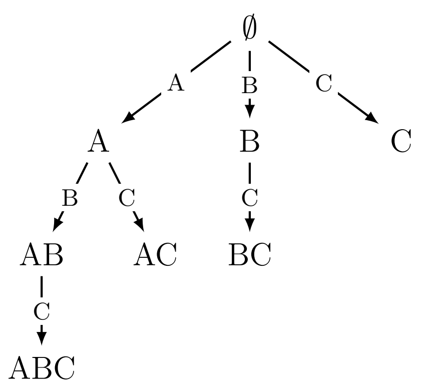
```
Input: Conjunctive Query Q
1:  push($OrderedQueue$,Q)
2:  while not $OrderedQueue$ is empty do
3:    SQ := pop($OrderedQueue$)
4:    PB := $\{\beta \in blocks(Q) \mid \pi(Q)^+ = ((\pi(Q) \cup \sigma(Q) \cup \beta)^+ \setminus (\sigma(Q) \cup \beta)^+)^+\}$
5:    if $\sigma$(SQ) = $\emptyset$ then
6:      $toadd$ := all blocks of PB
7:    else
8:      $\mathcal{KFK}$(Q) := ForeignKeyHandling(SQ)
9:      if not $\mathcal{KFK}$(SQ) then
10:       if monotonicty(SQ) then
11:         FQ := FQ $\cup$ GetFrequentInstances(SQ) //evaluate in database
12:         $support$(SQ) := $minsup$ //real support values in database
13:     if $support$(SQ) $\geq$ minsup then
14:       $uneq$ := all blocks of PB $\notin \sigma$(SQ)
15:       $toadd$ := all blocks $\beta$ in $uneq$ > last of $\sigma$(SQ)
16:   for all $\beta_i \in toadd$ do
17:     $\sigma$(SQC) := $\sigma$(CQ) with $\beta_i$ added
18:     $\sigma$(SQC) := closure of $\sigma$(CQC) under $\mathcal{FD}$(Q) $\cap$ PB
19:     if $\sigma$(SQC) $\notin Gen$ and $\sigma$(SQC) $\neq \pi$(Q) then
20:       push($OrderedQueue$, SQC)
21:       push($Gen$, $\sigma$(SQC))
22: return FQ
```

Selection Loop

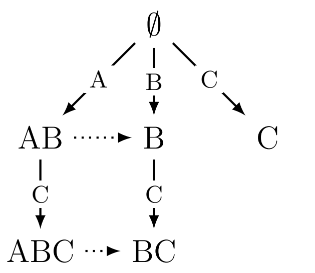
In the selection loop itself we also make use of the given (and discovered) functional dependencies. As in Conqueror, we generate new selection candidates by expanding the selection with new blocks. These blocks abide by the new constraints of closed queries (ensured on line 4 and 19 of Algorithm 3.9). Furthermore, in Conqueror$^+$, when adding blocks to the selection, in addition the closure is taken, ensuring no redundant queries are generated (line 18). However, closing of these sets of blocks requires us to reorder the queue of candidates in order to use the Apriori-trick. The following example illustrates this point.

Example 3.34. *Considering the attributes $\{A, B, C\}$, if no dependencies are present, generation of sets for the selection results in the generation-tree containing all levels shown in Figure 3.7a.*

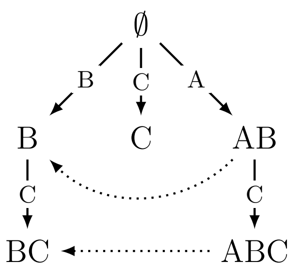
If the dependency $A \to B$ is present, we obtain the generation-tree in Figure 3.7b. Here, we notice that AB would be considered before B. However, because



(a) Selection tree of {A,B,C}



(b) Selection tree of {A,B,C} when $A \rightarrow B$ holds



(c) Reordered selection tree of {A,B,C} when $A \rightarrow B$ holds

Figure 3.7: Selection generation-trees of {A,B,C}

of the monotonicity property, we want to consider B before considering AB (because the selection according to B is less restrictive than that according to AB). We accomplish this by reordering the candidate queue, so as to ensure B is considered before AB and BC is considered before ABC, as illustrated in the generation-tree given in Figure 3.7c.

We note that the closure is computed using the standard algorithm [Ullman, 1988]. This algorithm adds to the set under construction X, the right handsides of those functional dependencies whose left handside is a subset of X, until no further changes are possible. Furthermore, in line 19 we make sure that the corresponding closure has not been processed previously, which can happen since a closed set can be generated from several non-closed sets. The instantiation of constant values from $\mathcal{I}$ (line 11) is performed analogously to Conqueror, *i.e.,* by performing SQL queries in the database, as explained in Section 3.2.2.

Algorithm 3.10 Functional Dependency Discovery

```
Input: PQ, FPQ
1: for all PQQ ∈ FPQ : $\exists\beta \in$ blocks$^+$(Q) such that $\pi$(PPQ) $\cup\ \beta = \pi$(PQ) do
2:   if support(PQ) = support(PPQ) then
3:     $\mathcal{FD}$(Q) := $\mathcal{FD}$(Q) $\cup \{\pi$(PQ) $\rightarrow \pi$(PPQ) $\setminus \pi$(PQ)$\}$
4:     mark PQ
```

3.6.2 Discovering Functional Dependencies

Based on Proposition 3.3, functional dependencies, other than those in $\mathcal{FD}$, are *discovered* in the projection loop (see line 18 of Algorithm 3.8) by calling Algorithm 3.10. At a given level of the lattice built up in this algorithm, a projection-join query PQ is generated from the projection-join queries PPQ of the previous level by removing one block β from $\pi(PPQ)$. Then, by Proposition 3.3, if *support*(PQ) = *support*(PPQ) (which is checked on line 2 of Algorithm 3.10), the instance $JQ(\mathcal{I})$ of the associated join query satisfies $\pi(PQ) \rightarrow \beta$. In this case, $\pi(PQ) \rightarrow \beta$ is added to $\mathcal{FD}_{JQ}$ and PQ is marked, since $\pi(JQ)$ is no longer closed under the new functional dependency (these two actions are performed on lines 3 and 4 respectively). These marked queries are then no longer considered for the evaluation of different constant values in the selection loop (line 32, Algorithm 3.9).

We note that such functional dependencies are discovered only if the queries PQ and PPQ having the same support are frequent (because, otherwise, PPQ has been pruned and its support cannot be retrieved). In other words, functional dependencies involving infrequent projections are not found. However, regarding frequent queries, every new functional dependency implies additional pruning in the selection loop, and in order to take full advantage, unlike the original Conqueror algorithm, the generation of selections is now performed after *all* projections are generated (line 31, Algorithm 3.9).

On the other hand, as projection-join queries are generated in a breadth-first manner, the 'best' functional dependencies (*i.e.,* those with minimal left-hand side) are discovered last, during the projection loop. Proceeding depth-first would allow us to discover the 'best' functional dependencies as early as possible, but at the cost of less monotonicity based pruning (see line 12, Algorithm 3.8). Moreover, by doing so, we can mark as many queries as possible in order not to process them in the selection loop. The following example illustrates this point.

Example 3.35. *In the context of sch*$(R_1) = \{A, B\}$ *and sch*$(R_2) = \{C, D, E\}$, *let us consider the projection loop where the join query that is considered is* $JQ = \pi_{ABCDE}\sigma_{(A=C)}(R_1 \times R_2)$. *In this case, blocks*$(JQ) = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$.

Assuming that all projections are frequent and that $JQ(\mathcal{I})$ *satisfies* $A \rightarrow D$, *the following dependencies are found:* $ACBE \rightarrow D$, $ACE \rightarrow D$, $ACB \rightarrow D$

and $AC \rightarrow D$. Consequently, the queries $\pi_{ACBE}(JQ)$, $\pi_{ACE}(JQ)$, $\pi_{ACB}(JQ)$ and $\pi_{AC}(JQ)$ are marked, because their supports are respectively equal to those of $\pi_{ACBED}(JQ) = JQ$, $\pi_{ACDE}(JQ)$, $\pi_{ACBD}(JQ)$ and $\pi_{ACD}(JQ)$. These queries are not processed by the selection loop.

We note that the functional dependency $A \rightarrow D$ is actually not found, because $\mathcal{FD}_{JQ}$ contains $A \rightarrow C$ and $C \rightarrow A$, which enforces A and C to appear together in the projections (this is a consequence of the fact A and C are in same block of blocks$^+(Q)$). Of course, $A \rightarrow D$ is a consequence of $AC \rightarrow D$ and $A \rightarrow C$ that now belong to $\mathcal{FD}_{JQ}$.

3.6.3 Discovering and Handling Foreign-Keys

Based on Proposition 3.6 we are able to discover inclusion dependencies. The first part of the discovery consists of finding the candidate keys of the relations. Since we already find functional dependencies, discovering a candidate key comes down to finding the attribute set on which all the other attributes depend. Of course it is possible that more than one attribute set qualifies as a candidate key.

The second part of the discovery is identifying the possible key-foreign-key joins. These are the join queries where we join a certain relation, R_k, with other relations (which we collectively refer to as R_{fk}) only on the key K of R_k. After identifying these types of joins, we need to verify if an inclusion dependency holds between the key of R_k and the attributes of another relation involved in the join, *i.e.* the possible foreign-key FK. As stated in Proposition 3.5, we can discover inclusion dependencies by means of query comparison. In order to do this, we must compare the support of the projection of the join query on the key-foreign-key attributes (K, FK), with the support of the projection of R_{fk} on the foreign-key (FK). We illustrate this in the following example:

Example 3.36. *Given the context of Example 3.35, and that we are considering the join query $J(Q) = \pi_{ABCDE}\sigma_{A=C}(R_1 \times R_2)$. Assume that we found that $A \rightarrow B$, and therefore have stored key$(R_1) = A$. Since we have a join of a key (A) with an attribute of another relation (C) this is a possible key-foreign-key join. To verify if we have an inclusion dependency, following Proposition 3.5, we need to evaluate support$(\pi_{ABC}\sigma_{A=C}(R_1 \times R_2))$ (note that B is included because of the key dependency) and compare it to the previously evaluated and stored support$(\pi_C(R_2))$. If the support values are equal we can conclude this is indeed a key-foreign-key join.*

This discovery is performed on line 5 of Algorithm 3.8, by means of Algorithm 3.11. Note that we only perform this discovery for join queries, since these are the first queries generated involving a new join. Algorithm 3.11 checks all individual joins (line 2) to see if the key of a relation is involved (line 5). If so, it constructs the query without this specific join (line 6), and compares the support

Algorithm 3.11 Foreign-Key Discovery

```
Input: Conjunctive Join Query JQ //JQ has the maximal projection for $\bowtie$(JQ)
1: $\mathcal{FK}(\text{JQ}) :=$ false
2: for all blocks $\beta \in \text{blocks(JQ)}$ where $\text{size}(\beta) > 1$ do
3:   for all $R$ in $\mathcal{D}$ do
4:     pkey := $\text{sch}(R) \cap \beta$
5:     if pkey = key(R) and $\forall \beta' \neq \beta : sch(R) \cap \beta' = \emptyset$ then
6:       $Q' :=$ JQ but where $\bowtie(Q') = (\bowtie(\text{JQ}) \setminus \{\beta\}) \cup \{(\beta \setminus \{key(R)\})\} \cup \{key(R)\}$
7:       if $support(\pi_{\beta}\text{JQ}) = support(\pi_{\beta \setminus key(R)} Q')$ then
8:         $\mathcal{FK}(\text{JQ}) :=$ true
9:         $\mathcal{KR}(\text{JQ}) := R$
10:        $\mathcal{FKJ}(\text{JQ}) := \bowtie(JQ) \setminus \{\beta\}) \cup \{(\beta \setminus \{key(R)\})\} \cup \{key(R)\}$
```

Algorithm 3.12 Foreign-Key Handling

```
Input: Conjunctive Query Q
Output: true only if redundant
1: JQ := join query associated with $\bowtie$(Q)
2: if $\mathcal{FK}(\text{JQ})$ then
3:   if $(\text{sch}(\mathcal{KR}(\text{JQ})) \setminus \text{key}(\mathcal{KR}(\text{JQ}))) \cap \sigma(\text{Q}) = \emptyset$ then
4:     if $\text{sch}(\mathcal{KR}(\text{JQ})) \subseteq \pi(\text{Q})$ or $\pi(\text{Q}) \cap \text{sch}(\mathcal{KR}(\text{JQ}))) = \emptyset$ then
5:       $\text{Q}' :=$ Q but where $\bowtie(\text{Q}') = \mathcal{FKJ}(\text{JQ})$ and $\pi(\text{Q}') = \pi(Q) \setminus \text{sch}(\mathcal{KR}(\text{JQ}))$
6:       if $\text{Q}' \in \text{FQ}$ then
7:         $support(\text{Q}) := support(\text{Q}')$
8:         link Q to instances Q'
9:       else
10:        $support(\text{Q}) := 0$ //infrequent
11:      return true
12: return false
```

values (line 7). If equal, Proposition 3.5 lets us conclude this specific join is a key-foreign-key join.

Having discovered such a key-foreign-key join, we can then immediately apply this knowledge to avoid unneeded query evaluations. The support of any projection on a subset of the attributes of R_{fk} (the foreign-key relation, or in general a join of relations) is equal to the support of the projection R_{fk} on the same attributes. Similarly, the support of any projection including the key of R_k is also known since it is the same as that of the projection of R_{fk} on the same attributes but with all R_k attributes removed. These conditions are checked on line 4 of Algorithm 3.12, which is called on line 10 of Algorithm 3.8 and line 8 of Algorithm 3.9. Additionally, for selections of these projections that do not contain any attributes from the

R_k relation (line 3), we also already know their support value. As explained in Section 3.4.5 we do not prune these queries, but we do not compute their support, instead retrieving it from the query that is equivalent with respect to the key-foreign-key join and that has been computed earlier (line 7).

3.6.4 Monotonicity and Association Rules

In Conqueror, queries involved in basic rules of the form $Q_1^+ \Rightarrow Q_2^+$ are those considered in the monotonicity check. In Conqueror$^+$ monotonicity and the basic rule generation are based on Proposition 3.10. We must note, however, that in the case of Conqueror$^+$ we cannot perform all monotonicity checks due to the nature of the generation procedure and the pre-ordering.

Example 3.37. *In the context of $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$, considering $CD \to E$ holds, then according to point 2 of Proposition 3.10*

$$\pi_{AC}\sigma_{(A=C)\wedge(B=b)}R \prec^1 \pi_{ACE}\sigma_{(A=C)\wedge(B=b)\wedge(D=d)}R$$

holds. However, it is clear that our current generation procedure will not have generated $\pi_{AC}\sigma_{(A=C)\wedge(B=b)}R$ when evaluating $\pi_{ACE}\sigma_{(A=C)\wedge(B=b)\wedge(D=d)}R$, as it has a more specific projection. As such, we cannot check its support, and we are also unable to compute the confidence of the basic rule

$$\pi_{AC}\sigma_{(A=C)\wedge(B=b)}R \Rightarrow \pi_{ACE}\sigma_{(A=C)\wedge(B=b)\wedge(D=d)}R$$

As an alternative, for the purpose of checking monotonicity, we can check the support of $\pi_{ACE}\sigma_{(A=C)\wedge(B=b)}R$, because this query has already been generated and it also holds that $\pi_{ACE}\sigma_{(A=C)\wedge(B=b)}R \prec \pi_{ACE}\sigma_{(A=C)\wedge(B=b)\wedge(D=d)}R$.

This limitation entails that we cannot fully exploit the monotonicity property in that case. We will, however, still consider these queries as input for the basic rule generation procedure, as at that stage all support values are known. Algorithm 3.13 details the new monotonicity procedure in Conqueror$^+$, taking dependencies and this limitation into account. As we can see on line 21 we check if the query is already generated, and otherwise consider the more general query. Note that in that case we do add the not yet generated query as a potential left hand side (line 20). The most notable change with respect to Conqueror is the consideration of all $X \subseteq \pi(Q)$ instead of just $\pi(Q)$ on lines 6 and 14 and the consideration of all $S \subseteq \sigma(Q)$ on line 12. These considerations together with the check for minimality on line 19, ensure that the conditions of Proposition 3.10 hold. The basic rule generation algorithm is identical to that of Conqueror, and based on the potential left hand side queries generated during monotonicity checking.

Algorithm 3.13 Conqueror$^+$ Monotonicity

```
Input: Conjunctive Query $Q$
1: for all blocks $\beta$ in $blocks^+(Q) \notin (\pi(Q) \cup \sigma(Q))$ do
2:   $\pi(Q') = ((\pi(Q) \cup \beta \cup \sigma(Q))^+ \setminus \sigma(Q))^+$; $\bowtie(Q') = \bowtie(Q)$; $\sigma(Q') = \sigma(Q)$
3:   $PP := PP \cup Q'$
4: for all blocks $\beta \in \sigma(Q)$ do
5:   $\sigma(Q') = \sigma(Q) \setminus \beta$; $\bowtie(Q') = \bowtie(Q)$
6:   for all $X \subseteq \pi(Q) : X = X^+$ and $\pi(Q) = ((X \cup \sigma(Q))^+ \setminus \sigma(Q))^+$ do
7:     $\pi(Q') = X$
8:     $SP := SP \cup \{Q'\}$
9: for all blocks $\beta$ in $blocks(Q)$ do
10:   for all all splits of $\beta$ in $\beta_1$ and $\beta_2$ do
11:     $\bowtie(Q') = (\bowtie(Q) \setminus \beta) \cup \{\beta_1, \beta_2\}$
12:     for all $S \subseteq \sigma(Q) : \beta \nsubseteq S$ and $S = S^+$ under $\mathcal{FD}(Q')$ and $S^+ = \sigma(Q)$
        under $\mathcal{FD}(Q)$ do
13:       $\sigma(Q') = S$
14:       for all $X \subseteq \pi(Q) : X = X^+$ under $\mathcal{FD}(Q')$ and
                $\pi(Q) = ((X \cup \sigma(Q))^+ \setminus \sigma(Q))^+$ under $\mathcal{FD}(Q)$ do
15:         $\pi(Q') = X$
16:         $JP := JP \cup \{Q'\}$
17: for all $MGQ \in (JP \cup PP \cup SP)$ do
18:   if $MGQ$ is not a cartesian product then
19:     if $\nexists Q' \in (JP \cup PP \cup SP) : MGQ \preceq Q'$ then
20:       $\mathrm{R}(Q) := \mathrm{R}(Q) \cup \mathrm{MGQ}$ //LHS for potential rule
21:       if $MGQ$ not generated then
22:         $MGQ := MGQ$ with $\pi(MGQ) = \pi(Q)$
23:       if $support(MGQ) < minsup$ then
24:         return false
25: return true
```

3.7 Conqueror$^+$ Experimental Results

We performed experiments using our new algorithm, Conqueror$^+$ and compared it to our previous version Conqueror. Again we look at QuizDB and IMDB as listed in Table 3.3.

The Conqueror$^+$ algorithm was written in Java using JDBC to communicate with an SQLite 3.4.0 relational database[4]. Experiments were run on a standard computer with 2GB RAM and a 2.16 GHz processor.

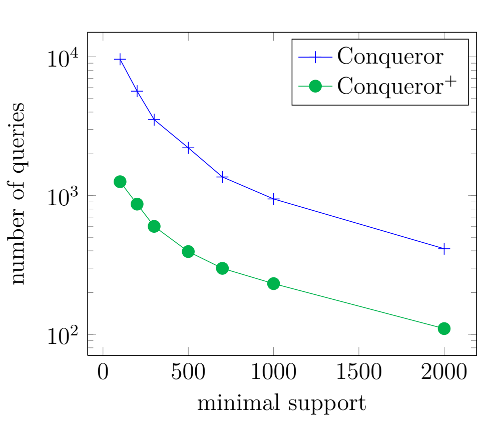
3.7.1 Impact of Dependency Discovery

As can be seen in Figure 3.8b, for the QuizDB, Conqueror$^+$ with discovery greatly outperforms Conqueror in runtime. The runtime for Conqueror$^+$ remains almost linear for a large portion of the support values, while for Conqueror without discovery it is increasing rapidly. This trend is mostly due to the large reduction in number of queries generated which is clearly shown in Figure 3.8a. For the IMDB we also have a reduction of redundant patterns as can be seen Figure 3.8c. The impact is, however, smaller, due to the small amount of attributes in the database. This reduces the impact that any use of functional dependencies can have on the results. It also results in the marginal improvement of runtime in IMDB (Figure 3.8d), since the largest component there is the computation of the join, which dependency based pruning has not much impact on.

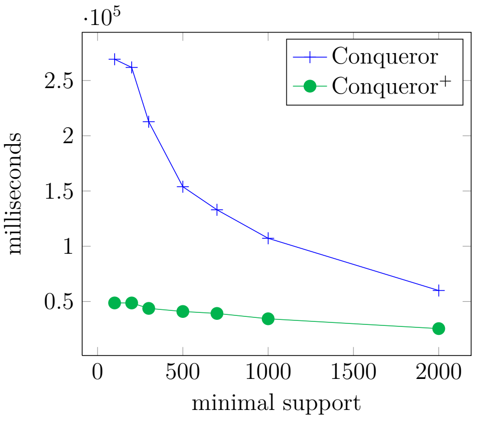
Next to this, we also performed an experiment to evaluate the effectiveness of dependency discovery. We performed an experiment on the QuizDB where we provided the key dependencies of the QUIZZES and SCORES relations, and disabled the discovery of new dependencies. As can be seen in Figure 3.8e the improvement with respect to Conqueror is marginal. To compare we performed a second experiment on QuizDB, this time without specifying any dependencies and enabling dependency detection. It is clear from Figure 3.8e that discovery on its own does result in a significant reduction of the number of patterns generated.

We also performed some time analysis to determine the cost and impact of functional dependency discovery. The results for an experiment using QuizDB are shown in Figures 3.10a and 3.10b. It is clear that the time needed for the discovery of functional dependencies (shown as ‘fdisc’ in Figure 3.10a) is negligible in comparison to the time gained in the selection loop (shown as ‘sel’). Adding discovery also requires extra time in the join loop (shown as ‘join’), but again, the gain in the selection loop outweighs this. If we look at the partitioning of time in Figure 3.10b, we clearly see that most time is spent in output and input. Since functional dependency discovery in Conqueror$^+$ greatly reduces output and

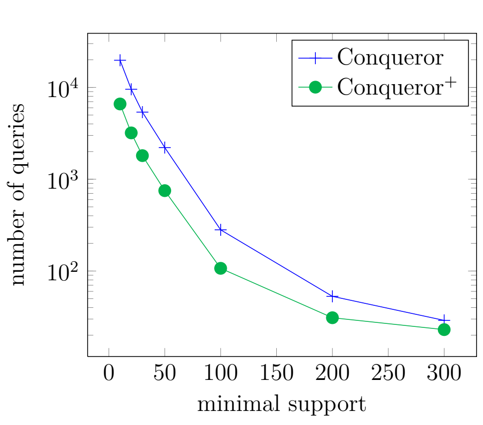
[4]The source code of Conqueror$^+$ can be downloaded at `http://www.adrem.ua.ac.be`.



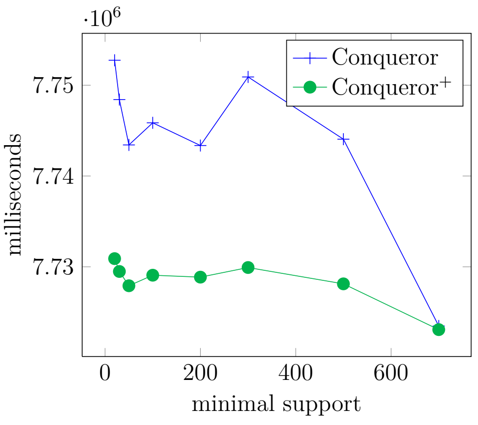
(a) QuizDB: number of patterns



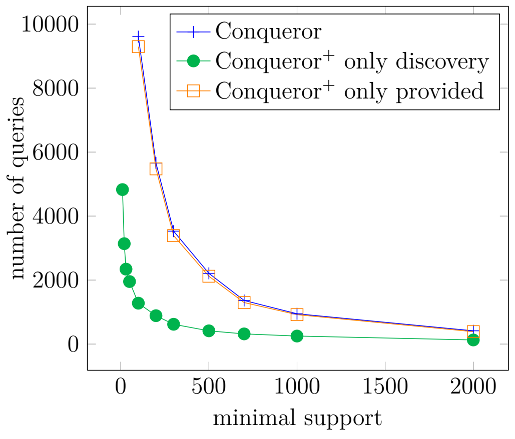
(b) QuizDB: runtime



(c) IMDB: number of patterns

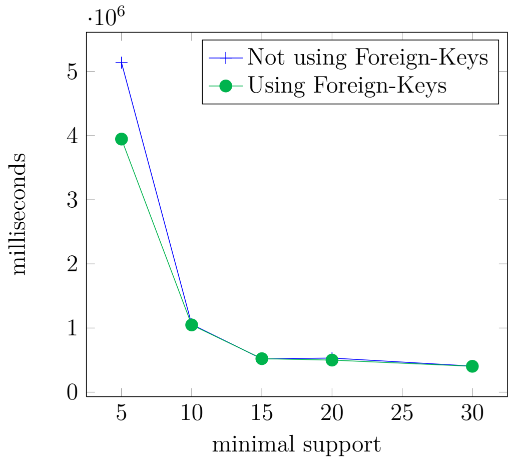


(d) IMDB: runtime

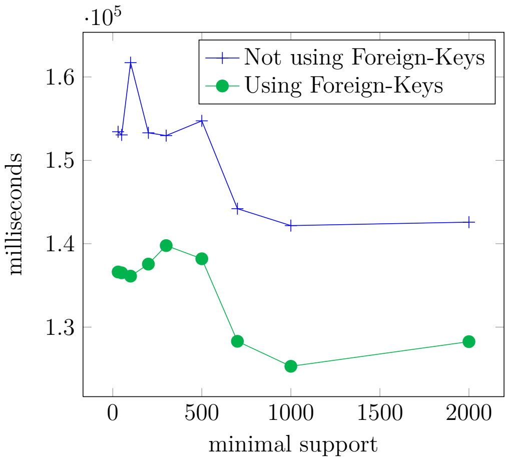


(e) QuizDB: Impact of discovering versus providing functional dependencies

Figure 3.8: Results for increasing support for Conqueror and Conqueror$^+$



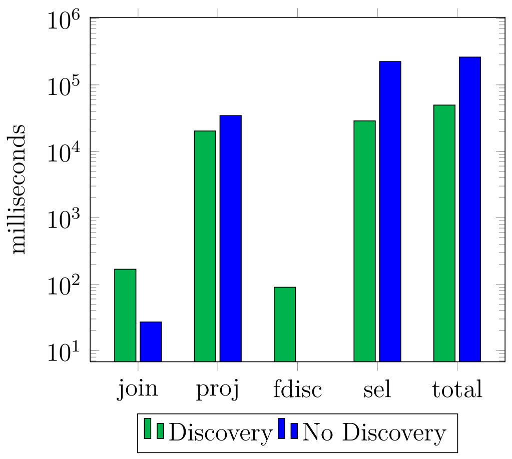
(a) **QuizDB:** Conqueror$^+$ runtime



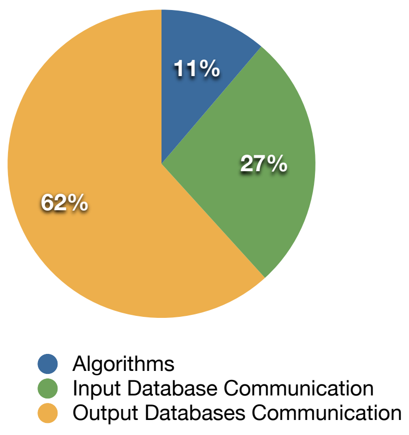
(b) **IMDB:** Conqueror$^+$ runtime

Figure 3.9: Results for increasing support for Conqueror$^+$

input by only considering queries Q^+, we get a large reduction in runtime as was observed in the experiments of Figure 3.8.



(a) Conqueror$^+$ Time Analysis of a QuizDB experiment



(b) Conqueror$^+$ Input/Output Time Analysis of a QuizDB experiment

Figure 3.10: Conqueror$^+$ Time analysis

To conclude we can state that discovering and subsequently using functional dependencies in conjunctive query mining provides the user with concise output and potentially significantly increases the performance.

3.7.2 Impact of Foreign-Keys

Unfortunately, the impact of Foreign-Keys is not so significant, as can be seen in Figure 3.9a. Only for a very low support threshold does the discovery and use of foreign-keys result in an increased performance. For higher support thresholds no significant difference can be observed. Investigating the results we see that the amount of queries whose evaluation is avoided is relatively small compared to the total amount of generated queries. This is firstly due to the fact that only one key-foreign-key join is present, and secondly due to the fact that there is only the small amount of attributes in the foreign-key-relation (*scores*) as well as functional dependencies between them. For the low support values the elimination of many constant-value queries does result in a positive runtime influence.

When looking at the results of an experiment on the IMDB database where we do not consider the costly join of *actormovies* and *genremovies*, shown in Figure 3.9b, a positive impact can be observed for all minimal support values. In the IMDB database multiple key-foreign-key joins are discovered. Although this results in the observed a overall improvement, the impact is also not very large in this case due to the small amount of attributes present in every relation.

Overall we can conclude that the structure and content of the database will determine the scale of impact of foreign-key usage. More experiments are required to investigate this issue further and are therefore part of future work.

3.7.3 Resulting Patterns

To illustrate the power of the patterns as well as the impact of functional dependencies, we list some results from the experiments on the QuizDB dataset.

As expected, making use of functional dependency discovery, we find the dependencies

$$\begin{gathered} quizzes.quizid \rightarrow quizzes.author, \\ quizzes.quizid \rightarrow quizzes.title, \\ quizzes.quizid \rightarrow quizzes.category, \\ quizzes.quizid \rightarrow quizzes.language \end{gathered}$$

As stated earlier, these functional dependencies are found next to various less optimal dependencies, which were discovered earlier.

The following rule was discovered with a confidence of 85%.

$$\pi_{\text{author,category,language}}\text{quizzes} \Rightarrow \pi_{\text{author,category}}\text{quizzes}$$

It states that knowing the author and category is enough to determine the language 85% of the time. This is an example of a *confident* functional dependency, as seen in Section 3.4.3. We are also able to find *conditional* functional dependencies (see Section 3.4.2) as 100% association rules:

$$\pi_{\text{author,title,category,language}}\sigma_{\text{category='misc'}}\text{quizzes} \Rightarrow \\ \pi_{\text{author,title,category}}\sigma_{\text{category='misc'}}\text{quizzes}$$

This rule states that for the tuples where attribute *category* equals 'misc', the dependency *author,title,category* $\rightarrow$ *language* holds.

With almost 100% (99.98%) the following rule holds

$$\pi_{\text{qid,title,year}}\text{scores} \Rightarrow \pi_{\text{qid,title,year}}\sigma_{\text{year='2006'}}\text{scores}$$

This tells us that the vast majority of quizzes were played in the year 2006.

With 91% confidence, the following rule states that most of the quizzes in the category 'history' have the language 'Dutch'. Also notice the fact that, using the dependency *quizzes.quizid* $\rightarrow$ *quizzes.author, quizzes.title, quizzes.category, quizzes.language*, all these attributes are present, and no redundant versions of this rule appear.

$$\pi_{\text{qid,author,title,cat,lang}}\sigma_{\text{cat='history'}}\text{quizzes} \Rightarrow \\ \pi_{\text{qid,author,title,cat,lang}}\sigma_{\text{cat='history',lang='Dutch'}}\text{quizzes}$$

The following rule is on the join of the quizzes and scores table. It states that 64% of the quizzes in category 'misc' that are played, are made by 'Damon'. Again, take note of the projection that is maximal under the found dependencies.

$$\pi_{\text{q.author,q.title,q.qid,q.cat,q.lang,s.year,s.name,scores.qid}}\ \sigma_{\text{q.qid=s.qid,q.cat='misc'}}\text{quizzes} \times \text{scores} \Rightarrow \\ \pi_{\text{q.author,q.title,q.qid,q.cat,q.lang,s.year,s.name,scores.qid}} \\ \sigma_{\text{q.qid=s.qid,q.cat='misc',q.author='Damon'}}\text{quizzes} \times \text{scores}$$

3.8 Related Work

Mining frequently occurring patterns in arbitrary relational databases has been the topic of several research efforts. [Dehaspe & Toivonen, 2001] developed the WARMR algorithm, that discovers association rules over datalog queries in an Inductive Logic Programming (ILP) setting. As stated in the introduction of this chapter, datalog queries correspond to conjunctive queries. Furthermore, we also stated that it is not feasible to consider all such (datalog) queries. To tackle this problem they introduce a declarative language bias. Such biases have been extensively studied in the field of inductive logic programming, where they require tight specification of patterns in order to consider huge, often infinite search spaces. The formalism introduced for Warmr is called Wrmode and is based on the Rmode format originally developed for the Tilde ILP system. This formalism requires two major constraints. The most important one is the *key constraint*. This constraint requires that a single *key* atomic formula is specified. This formula will then be obligatory in all queries. This key atomic formula essentially determines what is counted, *i.e.* the number of substitutions for the key variables with which the datalog query is true. Essentially this key describes what our transactions are going to be. The second constraint is the required *Atoms* list. This list contains all atomic formulas that are allowed in the queries that will be generated. In the most general case, this list consists of the relation names in the database schema $\mathcal{D}$. If one also wants to allow certain constants within the atomic formulas, then these atomic formulas must be specified for every such constant. In the most general case, the complete database instance must also be added to the *Atoms* list. Next to these basic requirements, additional constraints can be added using the Wrmode formalism, specifying exactly how the allowed queries can be formed (e.g. which bindings are allowed or the number of occurrences of an atom). These constraints dismiss a lot of potentially interesting patterns. We could find more, but this would require running the algorithm for every possible key atomic formula with the least restrictive language bias. In such a strategy, a lot of possible optimisations are left out. Furthermore, the query comparison used in Warmr is θ-subsumption [Plotkin, 1970], which is the inductive logic equivalent of conjunctive query containment [Ullman, 1989]. As we know from the introduction, checking containment and equivalence under this containment is an NP-complete problem, and this has an impact on the efficiency of the Warmr system. The candidate generation of Warmr starts off with the key query, the query only containing the key atomic formule. This is the most general pattern. Given a set of frequent queries $\mathcal{F}_i$ at a certain level i, Warmr generates a superset of all candidate patterns, by adding a single atomic formula, from the *Atoms* list, to every query in $\mathcal{F}_i$, following the Wrmode declarations. In a standard levelwise approach, for each such pattern, we check if all of its generalisations are frequent. In Warmr this is no longer possible, since some of these generalisations might not be allowed according to the specified language bias. Instead, Warmr scans all infrequent queries (such a list is kept) for a query that is more general than the considered potential candidate. This does, however, not imply that all queries that are more general than the considered query are indeed frequent.

Example 3.38. *Taking an example from [Goethals & Van den Bussche, 2002], consider the following two datalog queries that are single extensions of the key* *query (in this case visits(x_1, x_2)), and hence they are generated at the same level:*

$$\begin{aligned} Q_1(x_1, x_2) &:\!-\ \mathit{visits}(x_1, x_2), \mathit{likes}(x_1, \text{‘}\mathit{Duvel}\text{’}) \\ Q_2(x_1, x_2) &:\!-\ \mathit{visits}(x_1, x_2), \mathit{likes}(x_3, \text{‘}\mathit{Duvel}\text{’}) \end{aligned}$$

It is, however, clear that Q_2 is more general than Q_1. Both queries remain in the set of candidate queries, and even more, it is necessary that both queries remain, in order to guarantee that all queries are generated.

This example shows that the candidate generation of Warmr does not comply with the general levelwise framework given in Chapter 2. Furthermore, for each candidate query all other candidates and frequent queries are scanned for queries equivalent under θ-subsumption, which as stated before, is NP-complete. Moreover, the candidate evaluation of Warmr is performed within an inductive logic programming environment, evaluation of queries is therefore not that efficient. Warmr uses several optimizations to increase this performance, but this can hardly be compared to the optimised query processing in existing relational databases systems. The rule generation of Warmr just consists of finding all couples of queries (B, C) such that C θ-subsumes (contains) B. This is done in the straightforward brute-force way by checking all queries. However, we mention again that checking such containment cannot be done very efficiently.

[Nijssen & Kok, 2003a] introduce the Farmer algorithm, to tackle some of the efficiency issues present in Warmr. The pattern type considered is the same as Warmr and a mode mechanism similar to Wrmode, thus sharing the restrictions discussed above, is used. However, instead of using the inefficient logic programming based θ-subsumption, *Object Identity subsumption* (OI-subsumpsion) is used. In comparison to full θ-subsumption, using OI-subsumpsion makes the computation of equivalency slightly easier. It is, however, still a difficult problem, since it can be reduced to the graph isomorphism problem [Nijssen & Kok, 2003a]. The complexity of graph isomorphism is currently unknown. No polynomial algorithm exisits, nor does a proof of NP-completeness. Furthermore, Similar to Warmr, equivalent queries are removed by means of exhaustive search, and monotonicity is checked in a similar way, by means of storing all infrequent queries. We additionally note that Farmer does not include any rule generation.

[Goethals & Van den Bussche, 2002] studied a strict generalization of Warmr. They introduced the notion of *diagonal containment* as an extension to regular containment such that it could be better exploited in a levelwise algorithm. This allowed queries with essentially different key queries to be considered in the same search space. Unfortunately, this search space of all conjunctive queries is infinite and there exist no most general or even a most specific pattern, with respect to query containment and the defined frequency measure based on the number of tuples in the output of a query. [Goethals & Van den Bussche, 2002] apriori limited the search space to conjunctive queries with at most a fixed number of atomic formulas in the body. This way the set of most general queries is defined. Their candidate generation is based on applying operations on frequent queries (extension, join, selection, projection). Using these, they guarantee to generate all candidate queries, but several equivalent or redundant queries are also generated. Thus, equivalence testing must be applied in order to remove such queries. Unfortunately, deciding whether two conjunctive queries are equivalent under *diagonal* containment is *still* an NP-complete problem. In contrast to our approach [Goethals & Van den Bussche, 2002] only consider regular containment when generating association rules.

The efficiency issues of deciding containment and equivalence for conjunctive queries resulted in research focussed on smaller subclasses. A first special subclass of tree-shaped conjunctive queries, defined over a single binary relation representing a graph, was studied, showing that these tree queries are powerful patterns, useful for mining graph-structured data [Goethals et al., 2005, Hoekx & Van den Bussche, 2006]. [Jen et al., 2008] considered projection-selection queries over a single relation. They introduced a new notion of query equivalence, generalizing the standard one, based on comparisons of cardinalities, instead of being based on set inclusion and additionally taking functional dependencies into account. This work was a generalisation of previous work [Jen et al., 2006], considering the particular case of a database in which relations are organised according to a star schema. In this setting, frequent projection-selection queries are mined from the associated weak instance. However, in that setting, equivalence classes were defined based on projection only, and thus, only projection queries could be mined through one run of the proposed levelwise algorithm.

Other related approaches dealing with mining frequent queries also consider a fixed set of "objects" to be counted during the mining phase, but similar to Warmr only consider such objects to come from one relation for any given mining task [Kamber et al., 1997, Diop et al., 2002]. For instance, [Diop et al., 2002], characterise objects by a query, called the reference. Note that these approaches are also restricted to conjunctive queries, as in our case.

3.9 Conclusion

Conjunctive query mining research is motivated by the fact that many relational databases cannot always be simply transformed into transaction-like datasets in order to apply typical frequent pattern mining algorithms. As illustrated, possible transformations immediately strongly bias the type of patterns that can still be found, and hence, a lot of potentially interesting information gets lost. We have presented a new and appealing type of association rules, by pairing simple conjunctive queries. Next to many different kinds of interesting patterns, we have shown these rules can express *functional dependencies*, *inclusion dependencies*, but also their variants, such as the very recently studied *conditional functional dependencies*, which turn out to be very useful for data cleaning purposes. Moreover, association rules having a lower confidence than 100% are discovered, revealing so called *approximate dependencies*, which also have shown their merit in date cleaning. We presented a novel algorithm, Conqueror, capable of efficiently generating and pruning the search space of all simple conjunctive queries, and we presented promising experiments, showing the feasibility of our approach, but also its usefulness towards the ultimate goal of discovering patterns in arbitrary relational databases.

Then, we extended this basic approach to mine arbitrary relational databases, over which functional dependencies are assumed. Using functional dependencies, we are able to prune the search space by removing the redundancies they cause. Furthermore, since we are capable of detecting functional dependencies that were not given initially, we expanded our algorithm to also use these previously unknown functional dependencies to prune even more. Moreover, since not only the functional dependencies that hold on the database relations are discovered, but also functional dependencies that hold on joins of relations, these too are subsequently used to prune yet more queries. Besides functional dependencies, we also made our algorithm detect and use foreign-keys, since they can give rise to redundant query evaluations. We implemented and tested our updated algorithm, Conqueror$^+$, and we showed that it greatly outperforms the Conqueror algorithm by efficiently reducing the number of queries that are generated and evaluated, by means of detection and use of functional dependencies and foreign-keys. As such, the algorithms presented in this chapter provided an effective method for the discovery of a concise set of interesting and interpretable patterns in arbitrary relational databases.

3.10 Further Research

One of the open problems of the current Conqueror$^+$ algorithm is its inability to deal with mutually dependent attribute groups, as detailed in Example 3.33 in Section 3.6. In order to resolve this issue in the projection loop, when generating sub-projections, we need to consider the removal of more than one block at a time. The Algorithm 3.14 is meant to replace lines 21 to 29 in the projection loop given in Algorithm 3.8. This algorithm construct the set *torem*, consisting of sets of blocks that can be removed resulting in closed sub-projections. Since we are looking for the largest closed subsets of $\pi(PQ)$, we generate *torem* to contain the smallest subsets. This is achieved using levelwise generation on lines 11 to 15. On line 16 to 23 we essentially compute the (possibly singleton) sets of blocks that are mutually dependent with some other set of blocks. This knowledge is then used on line 1 to construct the new candidate blocks to be removed without reconsidering any blocks that have been removed before. Since sets of blocks can now be removed simultaneously, we need to keep track of the last removed set of blocks in order not to generate any duplicate candidates, this is achieved on line 23 and used on line 1.

Algorithm 3.14 New computation of candidate projections

```
1: $\mathit{candtorem} := \{\beta \in \pi(Q) \mid \beta > \min(\mathit{lastremoved}(\pi(PQ)) \wedge \beta > (\text{multiblock}(\beta)$
   $\cap\ \mathit{removed})\}$
2: if $\mathit{removed} \cap \mathit{dep} = \emptyset$ then
3:   $\mathit{candtorem} := \mathit{candtorem} \cup \mathit{dep}$
4: $\mathit{torem} := \emptyset$
5: while $\mathit{candtorem} \neq \emptyset$ and $\pi(PQ) \notin \mathit{candtorem}$ do
6:   for all $X \in \mathit{candtorem}$ do
7:     if $\pi(PQ) \setminus X$ is closed under $\mathcal{FD}(Q)$ then
8:       $\mathit{torem} := \mathit{torem} \cup \{X\}$
9:       $\mathit{candtorem} := \mathit{candtorem} \setminus \{X\}$
10:  $\mathit{newcandtorem} := \emptyset$
11:  for all $X \in \mathit{candtorem}$ do
12:    $k := \mathit{size}(X)$
13:    for all $Y \in \mathit{candtorem} : X[i] = Y[i]$ for $1 \leq i < k \wedge X[k] < Y[k]$ do
14:      $\mathit{newcandtorem} := \mathit{newcandtorem} \cup \{\{X \cup Y[k]\}\}$
15:  $\mathit{candtorem} := \mathit{newcandtorem}$
16: for all $X \in \mathit{torem}$ where $\text{size}(X) > 1$ do
17:   for all $Y \in \mathit{torem}$ where $\text{size}(Y) = \text{size}(X)$ do
18:     if $\text{size}(X \cap Y) = (\text{size}(X) - 1)$ then
19:       $\text{multiblock}(X \setminus (X \cap Y)) := \text{multiblock}(X \setminus (X \cap Y)) \cup \{Y \setminus (X \cap Y)\}$
20: for all $\beta_i \in \mathit{torem}$ do
21:   $\pi(PQC) := \pi(PQ)$ with block $\beta_i$ removed
22:   push($\mathit{Queue}$, PQC)
23:   $\mathit{lastremoved}(\pi(PQC)) := \beta_i$
```

This change to the projection generation algorithm should resolve the issues posed. The investigation of these changes, together with experimental evaluation, are subject of further research.

Next to this, as mentioned in Section 3.7.2, more experiments are required to further investigate the impact of using foreign-keys to reduce query evaluations.

Chapter 4

Relational Itemset Mining

ITEMSET MINING algorithms are probably the best known algorithms in frequent pattern mining. We introduced the problem and the basic techniques in Chapter 2. Many efficient solutions have been developed for this relatively simple class of patterns. These solutions can be applied to relational databases containing a *single* relation. Since a typical relation contains attributes that can have different values, this case defines an item as an attribute-value pair.

While the task of mining frequent itemsets in a single relation is well studied, for mining frequent itemsets in arbitrary relational databases, which typically have *more* than one relation, only a few solutions exist [Crestana-Jensen & Soparkar, 2000, Ng et al., 2002, Koopman & Siebes, 2008]. These methods consider a *relational itemset* to be a set of items, where each item is an attribute-value pair, and these items belong to one or more *relations* in the database. In order for two such items from different relations to be in the same itemset, they must be *connected*. Two items (attribute-value pairs) are considered connected if there exists a join of the two relations in the database that connects them. In general we can state that an itemset is valid if a tuple, containing the itemset, exists in the (full outer) join of all relations in the database. In this chapter we also adopt this definition.

A primary requirement to mine any kind of frequent patterns, is a good definition of a unit in which the frequency is expressed; that is, what is being counted. In standard itemset mining there is a single table of transactions and the unit to be counted is clearly defined as the number of transactions containing the itemset (See Section 2.1 of Chapter 2). For instance, a frequent itemset {*butter,cheese*}

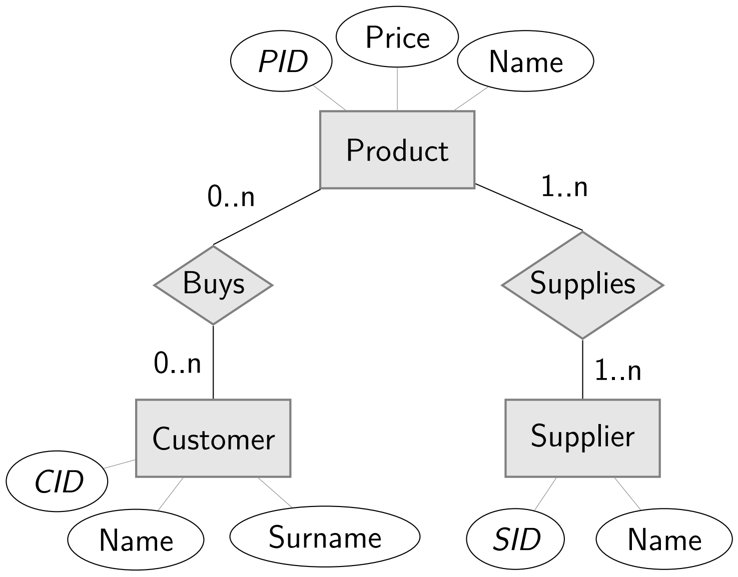


Figure 4.1: Supermarket Database

represents the fact that in a large fraction of the transactions *butter* and *cheese* occur together. When considering itemsets over multiple relations, elegantly defining such a unit is less obvious in general. In the existing relational itemset mining approaches [Crestana-Jensen & Soparkar, 2000, Ng et al., 2002, Koopman & Siebes, 2008], the frequency of an itemset over multiple relations is expressed in the number of occurrences (tuples) in the result of a join of the relations in the database. However, this definition of frequency of an itemset is hard to interpret as it heavily depends on how well the items are connected to other items (not necessarily in that itemset). For example, consider the relational database with the scheme represented in the Entity-Relationship model in Figure 4.1. Then consider the singleton itemset {Product.Name=*butter*}. In the join of Customer, Product and Supplier, this itemset could occur frequently because it is connected to a lot of *customers*, but even if this is not the case, it could still be frequent simply because it is connected to a lot of *suppliers*. The main issue here is that when using the number of occurrences in the join as a frequency measure, it is hard to determine the true *cause* of the frequency.

Instead of counting frequency as the number of occurrences in the join, we opt for a novel, more semantic approach. In the supermarket example, we want to find products that are bought by a lot of (different) customers, but we might also be interested in products that have a lot of (different) suppliers. Essentially, we want to count the number of connected customers and the number of connected suppliers. In order to have such semantics, we must not count all occurrences of

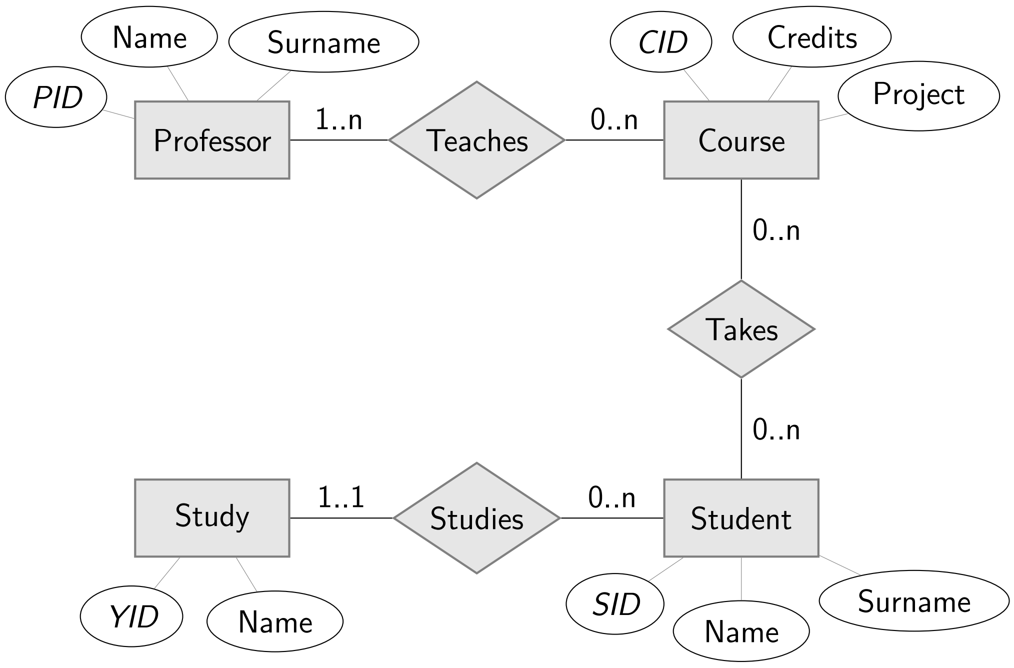


Figure 4.2: Running Example Relational Scheme

an itemset in the join, but instead separately count the occurrences with unique customers and the occurrences with unique suppliers. This new frequency counting technique allows for interpretable frequent itemsets, since we will now have separate customer-frequent and supplier-frequent itemsets.

As our practical goal is to mine relational itemsets in arbitrary relational databases, we assumed key-dependencies are specified with the relational scheme of the input database. This way we can count unique key values, as a way of counting connected entities. Consider the example Entity-Relationship scheme in Figure 4.2, which we will use as a running example throughout the chapter. For this scheme, the keys to be used would be in the set {Professor.PID, Course.CID, Student.SID, Study.YID}. It goes without saying that itemsets frequent in Professor.PID have a different semantics than itemsets frequent in Course.CID. It is clear that our new frequency counting approach significantly differs from other relational itemset mining approaches that simply count the occurrences in the join of all relations. As we show in this chapter, our new approach allows for an efficient propagation based depth-first algorithm that generates interesting frequent relational itemsets that are sensible and easily interpretable. As stated before, we want to provide a pattern mining algorithm that can directly interact with standard *relational databases*. We therefore use SQL in our implementation to fetch the necessary data from the database. As a result, our prototype is generally and easily applicable for arbitrary relational databases, without requiring any transformation on the data.

We formally define relational itemsets, rules and our novel frequency counting approach in Section 4.1. Two variants of a depth-first algorithm for mining frequent relational itemsets are proposed in Section 4.2. Section 4.3 investigates a new measure based on deviation. We then investigate several types of association rule redundancy, in order to reduce the size of the output in Section 4.4. In Section 4.5 we consider the results of experiments comparing the variations of the algorithm and investigate the patterns found. Finally, we consider related work in Section 4.6 and conclude in Section 4.7. Part of this chapter is based on work published in [Goethals et al., 2009, Goethals et al., 2010].

4.1 Definitions

Before formally defining relational itemsets, we first consider the relational scheme as it forms the basis of our definition of patterns.

4.1.1 Relational Scheme

Every relational database has a relational scheme (see Chapter 1). For clarity, we focus on simple relational schemes. More specifically, using the Entity-Relationship model, we consider acyclic relational schemes using only binary relations to connect entities, *i.e.*, schemes that can be represented as unrooted trees (See Figure 4.2 as an example). Let *sort* be a function that maps a relation name to its attributes [Abiteboul et al., 1995]. We now define such schemes as follows:

Definition 4.1. *Let $\mathcal{E}$ be a set of entities and $\mathcal{R}$ a set of binary relations. A* ***simple relational scheme*** *is a tuple $(\mathcal{E}, \mathcal{R})$ such that*

1. $\forall E \in \mathcal{E} : \exists! key(E) \subseteq sort(E)$, *the key attributes of* E
2. $\forall R \in \mathcal{R} : \exists! E_i, E_j \in \mathcal{E}, E_i \neq E_j$ *such that* $sort(R) = key(E_i) \cup key(E_j)$
3. $\forall E_i, E_j \in \mathcal{E}, E_i \neq E_j : \exists! E_1, \ldots, E_n \in \mathcal{E}$ *such that*
   a) $E_1 = E_i$ *and* $E_n = E_j$
   b) $\forall k, l :$ *if* $k \neq l$ *then* $E_k \neq E_l$
   c) $\forall k, \exists! R \in \mathcal{R} : sort(R) = key(E_k) \cup key(E_{k+1})$

Informally, this states that every entity needs to have a unique key defined, and that for every two entities there exists a unique path of binary relations and entities connecting them. Many practical relational databases satisfy such simple relational schemes. Moreover, databases with star schemes and snowflake schemes can be formulated in this way.

4.1.2 Relational Itemsets

We are now ready to formally define the type of pattern we want to mine.

Definition 4.2. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, $\{(A_1 = v_1), \ldots, (A_n = v_n)\}_K$ is a **relational itemset** in key K where $K \in \bigcup_{E \in \mathcal{E}}\{key(E)\}$, each $(A_i = v_i)$ is an attribute-value pair (or item) such that $A_i \in \bigcup_{E\in\mathcal{E}} sort(E)$ and $v_i \in Dom(A_i)$ and $\nexists(A_j, v_j) : j \neq i$ with $A_j = A_i$. We denote the set of all items by $\mathcal{I}$.*

Note, that we will sometimes use I to denote a set of items without any key specified, next to I_K which denotes an itemset in key K.

Example 4.1. *Given the relational scheme in Figure 4.2, where we abbreviate the relation names to P, C, Stnt and Stdy, we can consider the relational itemset $\{(\mathsf{P.Name} = \mathsf{Jan}), (\mathsf{C.Credits} = \mathsf{10})\}_{\mathsf{C.ID}}$. Suppose this relational itemset is frequent, then it expresses that a large fraction of the courses has 10 credits and that a professor that teaches them is named Jan. Since we have C.ID as the counted key we know we are expressing patterns about courses. Because we assume a simple relational scheme, the professor can only be connected to the courses in one way (via the relation Teaches) and hence, we know this pattern expresses that Jan is teaching them.*

Next, we define the frequency measure *support* for relational itemsets. In order to do this we need to consider the unique path of entities and relations connecting the entities of the itemset.

Proposition 4.1. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, and a relational itemset $I_K = \{(A_1 = v_1), \ldots, (A_n = v_n)\}_K$, let $\mathcal{E}_{I_K} = \{E \in \mathcal{E} \mid \text{key}(E) = K \text{ or } \exists i : A_i \in sort(E)\}$. There exists a **unique path** P_{I_K} connecting all $E \in \mathcal{E}_{I_K}$.*

Proof. (outline) We can consider the simple relational scheme $(\mathcal{E}, \mathcal{R})$ to be a tree where $\mathcal{E}$ are the nodes, and $\mathcal{R}$ are the edges. Then we can consider the subtree formed by the nodes in $\bigcup_{E_i, E_j \in \mathcal{E}_{I_K}} \text{path}(E_i, E_j)$ and the edges in $\mathcal{R}$ connecting them, where $\text{path}(E_i, E_j)$ is the unique path of entities as defined in Definition 4.1. If we take $E_K : \text{key}(E_K) = K$ to be the root of this tree, we can then consider the preorder traversal of this subtree as the unique path P_{I_K}. □

Definition 4.3. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$ the **absolute support** of a relational itemset $I_K = \{(A_1 = v_1), \ldots, (A_n = v_n)\}_K$ is the number of distinct values of the key K in the answer of the query (expressed here in relational*

Professor

PID	Name	Surname
A	Jan	P
B	Jan	H
C	Jan	VDB
D	Piet	V
E	Erik	B
F	Flor	C
G	Gerrit	DC
H	Patrick	S
I	Susan	S

Course

CID	Credits	Project
1	10	Y
2	10	N
3	20	N
4	10	N
5	5	N
6	10	N
7	30	Y
8	30	Y
9	10	N
10	10	N
11	10	N

Student

SID	Name	Surname
1	Wim	LP
2	Jeroen	A
3	Michael	A
4	Joris	VG
5	Calin	G
6	Adriana	P

Study

YID	Name
I	Computer Science
II	Mathematics

Studies

SID	YID
1	I
2	I
3	I
4	II
5	II
6	II

Teaches

PID	CID
A	1
A	2
B	2
B	3
C	4
D	5
D	6
E	7
F	8
G	9
G	10
G	11
I	11

Takes

SID	CID
1	1
1	2
2	1
3	1
4	3
5	2
6	11

Figure 4.3: Running Example Relational Instance

algebra [Abiteboul et al., 1995]):

$$\pi_K \sigma_{A_1=v_1,\ldots,A_n=v_n} E_1 \bowtie_{\text{key}(E_1)} R_{1,2} \bowtie_{\text{key}(E_2)} E_2 \bowtie \cdots \bowtie E_n$$

*where $E_i \bowtie_{\text{key}(E_i)} R_{i,i+1}$ represents the equi-join on $E_i.\text{key}(E_i) = R_{i,j}.\text{key}(E_i)$ and all $E_i \in \mathcal{E}_{I_K}$ are joined using the unique path P_{I_K}. The **relative support** of an itemset is the absolute support of the itemset divided by the number of distinct values for K in the entity of which K is key. We call a relational itemset **frequent** if its (absolute/relative) support exceeds a given* minimal (absolute/relative) support *threshold.*

Example 4.2. *Given the instance of the scheme of Figure 4.2 depicted in Figure 4.3, the absolute support of the itemset $\{(\mathsf{C.Credits} = \mathsf{10})\}_{\mathsf{P.PID}}$ is 6, as the*

tid	P.PID	P.Name	P.Surn	C.CID	C.Project	C.Credits	S.SID	S.Name	S.Surn	Y.YID	Y.Name
1	A	Jan	P	1	Y	10	1	Wim	LP	I	Comp. Sc.
2	A	Jan	P	1	Y	10	2	Jeroen	A	I	Comp. Sc.
3	A	Jan	P	1	Y	10	3	Michael	A	I	Comp. Sc.
4	A	Jan	P	2	N	10	1	Wim	LP	I	Comp. Sc.
5	A	Jan	P	2	N	10	5	Calin	G	II	Math.
6	B	Jan	H	2	N	10	1	Wim	LP	I	Comp. Sc.
7	B	Jan	H	2	N	10	5	Calin	G	II	Math.
8	B	Jan	H	3	N	20	4	Joris	VG	II	Math.
9	C	Jan	VDB	4	N	10	NULL	NULL	NULL	NULL	NULL
10	D	Piet	V	5	N	5	NULL	NULL	NULL	NULL	NULL
11	D	Piet	V	6	N	10	NULL	NULL	NULL	NULL	NULL
12	E	Erik	B	7	Y	30	NULL	NULL	NULL	NULL	NULL
13	F	Flor	C	8	Y	30	NULL	NULL	NULL	NULL	NULL
14	G	Gerrit	DC	9	N	10	NULL	NULL	NULL	NULL	NULL
15	G	Gerrit	DC	10	N	10	NULL	NULL	NULL	NULL	NULL
16	G	Gerrit	DC	11	N	10	6	Adriana	P	II	Math.
17	H	Patrick	S	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
18	I	Susan	S	11	N	10	6	Adriana	P	II	Math.

Figure 4.4: Full outer join of the relational instance of Figure 4.3

distinct answer to the query

$$\pi_{P.PID}\sigma_{C.Credits=10} P \bowtie_{PID} \mathit{Teaches} \bowtie_{CID} C$$

is {A,B,C,D,G,I}. This means that six professors teach a course with 10 credits. The relative support is 6/9, as there are 9 professors in the Professor *relation.*

4.1.3 Relational Association Rules

Association rules are defined in much the same way as in standard itemset mining (see Chapter 2). The only restriction added in the setting of relational itemsets, is that the antecedent and the consequent need to be expressed in the same key.

Definition 4.4. *Let $\mathcal{I}_K$ be the set of all relational itemsets in key K. $A \Rightarrow_K C$ is a* ***relational association rule*** *in key K if $A, A \cup C \in \mathcal{I}_K$.*

Definition 4.5. *The* ***support*** *of $A \Rightarrow_K C$ is the support of $(A \cup C)_K$. The* ***confidence*** *of $A \Rightarrow_K C$ is the support of $(A \cup C)_K$ divided by the support of A_K.*

Example 4.3. *Given the instance depicted in Figure 4.3, a possible relational association rule could be $\{(P.Name = Jan)\} \Rightarrow_{C.CID} \{(C.Credits = 10)\}$ The confidence is 3/4 = 0.75 since there are 3 courses (1,2,4) taught by a 'Jan' that have 10 credits, compared to the 4 courses (1,2,3,4) taught by a 'Jan' in total. The relative support is 3/11 = 0.27 since there are 11 courses in total.*

4.2 Algorithm: SMuRFIG

In this section we describe several algorithms for mining relational itemsets. We first consider an approach for a naive algorithm, based on the computation of the full outer join. Then we present our algorithm **SMuRFIG** (**S**imple **Mu**lti-**R**elational **F**requent **I**temset **G**enerator). All algorithms employ *KeyID lists*, similar to the transaction identifier (*tid*) lists used in the well-known Eclat algorithm (see Chapter 2). The KeyID list of an itemset I_K is defined as the set of distinct values in the answer of the support query specified in Definition 4.3.

4.2.1 Naive Relational Itemset Miner

First, we consider the naive approach. The input of the Naive algorithm (see Algorithm 4.1) is an instance of a simple relational scheme as defined in Section 4.1 and a relative minimum support threshold *minsup*. The support query from Definition 4.3 is straightforwardly decomposed into three parts, *i.e.* a join, a selection, and a projection. First, a join table is constructed using entities and relations, and then the correct support(s) are found using this table. However, this join is different for each itemset, and we would have to perform all such possible joins, which is infeasible. Instead, we create a single large join table containing all entities and relations. To achieve this, we cannot use an equi-join. Indeed, if a tuple is not connected to any tuples in other tables, it would not appear in the full equi-join of all entity tables, which means we would lose some information. To avoid this, we combine the entities and relations using a *full outer join*, which combines all non-connected tuples with NULL-values. This full outer join results in table J on line 1.

Example 4.4. *The result of a full outer join of the entities and relations of Figure 4.3 is given in Figure 4.4. Here, there are* NULL *values present for the student attributes on the rows for courses 4 to 10, since no students are taking them.*

The minimum support threshold is relative to each entity separately, hence we cannot simply use it relative to the join table J. Instead, we must use an absolute minimum support threshold *abssup* (line 2) as a lower bound, since the absolute support of an itemset in J is at least as high as the absolute support of the itemset in any entity table E, so any itemset frequent in some E with respect to *minsup* will also be frequent in J with respect to *abssup*. A standard frequent set miner, Eclat, is executed on table J, using the new threshold (line 3). This stage of the algorithm effectively performs the select clause (σ) of the support query. We assume here that the tid lists of an itemset (the list of tuples of J where the itemset occurs) are part of the output of Eclat. Then, the tid lists of all generated itemsets are *translated* to their appropriate KeyID lists on line 6. Translating a tid list T

Algorithm 4.1 Naive Relational Itemset Miner

```
Input: An instance of a simple relational scheme $(\mathcal{E}, \mathcal{R})$; relative support threshold minsup
Output: Set $\mathcal{F}$ of all itemsets $I$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is frequent
1: $J := E_1 \bowtie R_{1,2} \bowtie E_2 \bowtie \cdots \bowtie E_n$
2: abssup := $\min_{E \in \mathcal{E}}$(minsup $\times |E|$)
3: $\mathcal{I}_t$ := Eclat($J$, abssup)
4: for all $I_t \in \mathcal{I}_t$ do
5:   for all $E \in \mathcal{E}$ do
6:     KeyIDs($I_{key(E)}$) := translate($I_t$,key($E$))
7:     if $support(I_{key(E)}) \geq$ minsup $\times |E|$ then
8:       $\mathcal{F} := \mathcal{F} \cup I_{key(E)}$
9: return $\mathcal{F}$
```

to a KeyID list is comes down to performing the projection $\pi_K(J \bowtie T)$ to each key K. This can be done efficiently by using a lookup table that can be created during the construction of J. Finally, the relative minimum support threshold is imposed for each itemset, and the frequent itemsets are reported.

Example 4.5. *The itemset $\{(\mathsf{C.Credits} = \mathsf{10})\}$ corresponds to the transaction ids {1,2,3,4,5,6,7,9,11,14,15,16,18} of the full outer join shown in Figure 4.4. If we translate these to unique* P.PID*s by looking up the* P.PID *values for these tuple ids in the join table, we get {*`A,B,C,D,G,I`*}, and hence the absolute support of itemset $\{(\mathsf{C.Credits} = \mathsf{10})\}_{\mathsf{P.PID}}$ is 6, corresponding to the results in Example 4.2.*

The advantage of this naive approach is that it can be implemented as a wrapper around an existing itemset mining algorithm. However, there are several drawbacks to this naive method. First of all, the computation of the full outer join is costly with respect to both computation time and memory consumption. While still possible for small databases, it quickly becomes infeasible for larger ones. Secondly, it generates a lot of candidates. We can only prune itemsets that are infrequent in J with respect to *abssup*. However, many candidate itemsets that are frequent in J, will turn out to be infrequent in all entity tables E with respect to the *minsup* threshold.

4.2.2 SMuRFIG

The SMuRFIG algorithm (see Algorithm 4.2) does not suffer from these drawbacks, *i.e.* it is fast and efficient in both time and memory. It uses the concept of *KeyID list propagation.* First, the KeyID lists of singleton itemsets are extracted from the data in their respective entities (line 5), and then these KeyID lists are propagated to all other entities (line 7, see Algorithm 4.4). The propagation function recursively translates a KeyID list from one entity E_i to its adjacent entities E_j, until all entities are reached. The translation of a KeyID list from E_i to E_j using $R_{i,j}$ is equivalent to the result of the relational query $\pi_{key(E_j)}(KeyIDs_{E_i} \bowtie R_{i,j})$. It is not difficult to verify that in this way we now have the KeyID lists of all items for all entities E, and hence their supports. Next, the (frequent) singleton itemsets are combined into larger sets by the *Keyclat* function (Algorithm 4.3).

The input of SMuRFIG is the same as the naive algorithm. SMuRFIG also first calculates the minimal absolute support, but this time it is to be used as a general threshold to prune singleton itemsets in each entity table separately. Furthermore, as an optimisation, each entity table is subject to a left outer join ($\lhd$ on line 4) in SQL. A left outer join is similar to a full outer join, but will only always keep all attributes from the *left* relation and adds the attributes from the right relation. Unlike the full outer join, a left outer join can be efficiently computed. Projecting this outer join back on the attributes of the leftmost entity, assuming no duplicates are eliminated, has the effect of "blowing-up" the occurrences of the attributes of this entity. The attributes now occur in the same quantity as they would in the full outer join and this allows us to apply the computed absolute minimum support threshold when computing the singleton itemsets. If we would not blow-up the entity tables, some attribute values could be underrepresented, and we would not be able to prune at all in the singleton mining phase. That is to say, a certain attribute could be infrequent in the key of its own entity (because it only appears in a small number of tuples), but this same attribute can, however, be frequent in a key of a different entity than the one the attribute belongs to. This is the case

Algorithm 4.2 SMuRFIG

```
Input: An instance of a simple relational scheme $(\mathcal{E}, \mathcal{R})$; relative support threshold $minsup$
Output: Set $\mathcal{F}$ of all itemsets $I$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is frequent
 1: abssup := $\min_{E \in \mathcal{E}}(\text{minsup} \times |E|)$
 2: for all $E \in \mathcal{E}$ do
 3:   $K := key(E)$
 4:   $E' := \pi_{\text{sort}(E)} E \lhd R_1 \lhd \cdots \lhd R_{|\mathcal{R}|}$
 5:   $\mathcal{I}_E$ := singletons($E'$, abssup)
 6:   for all $I_K \in \mathcal{I}_E$ do
 7:     Propagate(KeyIDs($I_K$))
 8:     if $\exists K' : support(I_{K'}) \geq minsup \times |E|$ then
 9:       $\mathcal{I} := \mathcal{I} \cup \{I\}$
10: $\mathcal{F}$ := Keyclat($\mathcal{I}$, minsup)
11: return $\mathcal{F}$
```

when this small number of tuples in the entity are connected to a lot of tuples in another entity.

Example 4.6. *Considering the instance of Figure 4.3. Here the itemset {(P.Name = `Gerrit`)} only occurs once in the Professor table. Thus, counting professors, this attribute is infrequent for an absolute minimal support of 2. But if we look at the full outer join of Figure 4.4 we see {(P.Name=`Gerrit`)} appears 3 times, because '`Gerrit`' is connected to 3 courses (9,10,11). Counting courses, {(P.Name = `Gerrit`)} should be frequent for the absolute minimal support of 2. After performing a left outer join on the entity Professor, the item {(P.Name=`Gerrit`)} will appear 3 times, and will therefore now be frequent with the same absolute minimal support in this new blown-up entity.*

Because of the left outer join, we can now safely prune infrequent items (using the computed absolute minimal support threshold) within each blown-up entity, since we know that, due to the blow-up, infrequent items are infrequent in any key. The actual mining of the singletons is performed by the subroutine *singletons* (line 5). It just scans the table, extracts the frequent singleton items, and prunes the infrequent ones. The support count used for this pruning is based on the row-ids (or transaction-ids) of the (blown-up) entity table. This is therefore not the same as the KeyID based support used in the rest of the algorithm. After this initial pruning in the singleton mining phase we compute the support of a singleton attribute in the key of its entity, basically by eliminating duplicates. Thus for each entity E we now have all singleton itemsets expressed in $key(E)$, where each singleton contains one attribute of E (line 5).

We then propagate these KeyID lists to all other keys (line 7). From this point on we only use KeyID based support, and, using the singleton itemsets as input, all frequent itemsets are mined in the subroutine *Keyclat* (Algorithm 4.3), which uses the provided relative minimal support threshold.

The Keyclat routine is the core of SMuRFIG. It traverses the search space depth-first, and uses the concept of *KeyID propagation* to correctly and efficiently compute KeyID lists of an itemset. In each recursion, two k-itemsets with a common prefix P (initially empty) are combined to form a new candidate set of size $k + 1$. Let I' and I'' be two such itemsets, and let $I = I' \cup I''$. To compute the support of I, we first determine that the entity tables of the suffix items of I' and I'' are E_1 and E_2 (line 4). We then intersect the KeyID lists of I' and I'' in E_1 and E_2, to obtain the support of I in E_1 and E_2. Then, these KeyID lists are *propagated* to all other entities (line 8). For all other entities E we now intersect these propagated KeyID lists with the KeyID lists of I' and I'', to obtain the KeyID list of I in E (line 12). In some cases, however, it is not necessary to make the intersection on line 12. If an entity E does not lie *between* E_1 and E_2 in the database's scheme graph (which is the case if the unique path from E to E_1

Algorithm 4.3 Keyclat

```
Input: Set of $k$-itemsets $\mathcal{L}_P$ with a common prefix $P$; support threshold $minsup$
Output: Set $\mathcal{F}$ of all itemsets $I$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is frequent and with common prefix $P$
1: for $I'$ in $\mathcal{L}_P$ do
2:   for $I''$ in $\mathcal{L}_P$ with $I'' > I'$ do
3:     $I := I' \cup I''$
4:     $E_1, E_2 :=$ entity of suffix items $I' \setminus P, I'' \setminus P$ respectively
5:     for $i \in \{1, 2\}$ do
6:       $K_i := key(E_i)$
7:       KeyIDs$(I_{K_i}) :=$ KeyIDs$(I'_{K_i}) \cap$ KeyIDs$(I''_{K_i})$
8:       pKeyIDs$_i(I) :=$ Propagate(KeyIDs$(I_{K_i})$)
9:     for $E \in \mathcal{E} \setminus \{E_1, E_2\}$ do
10:      $K := key(E)$
11:      if $E$ lies between $E_1$ and $E_2$ then
12:        KeyIDs$(I_K) :=$ pKeyIDs$_1(I_K) \cap$ pKeyIDs$_2(I_K)$
                          $\cap$ KeyIDs$(I'_K) \cap$ KeyIDs$(I''_K)$
13:      else
14:        $E_i$ is the closest to $E$
15:        KeyIDs$(I_K) :=$ pKeyIDs$_i(I_K)$
                          $\cap$ KeyIDs$(I'_K) \cap$ KeyIDs$(I''_K)$
16:      $support(I_K) := |$KeyIDs$(I_K)|$
17:      if $support(I_K) \geq$ minsup $\times |E|$ then
18:        $\mathcal{F}_{I'} := \mathcal{F}_{I'} \cup I_K$
19:   $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_{I'} \cup$ Keyclat$(\mathcal{F}_{I'}$, minsup$)$
20: return $\mathcal{F}$
```

Algorithm 4.4 Propagate

```
Input: KeyID list of an itemset $I$ in some $key(E_i)$
Output: KeyID lists of $I$ in all $key(E)$ for $E \in \mathcal{E}$
1: for all neighbours $E_j$ of $E_i$ not yet visited do
2:   Translate KeyIDs in $key(E_i)$ to $key(E_j)$ using $R_{i,j}$
3:   Propagate(KeyIDs in $key(E_j)$)
```

contains E_2 or vice versa), it is sufficient to take the propagated KeyID list of the entity E_1 or E_2 closest to E in the scheme's graph (line 15).

Example 4.7. *Suppose that we have the two itemsets $P = \{(P.Name = Jan)\}$ and $S = \{(S.Surname = A)\}$. These are singletons, so we have computed the KeyID lists for all keys. Furthermore, their common prefix is the empty set. We determine* *the entity table of P to be* Professor *and of S to be* Student*. Let PS denote $P \cup S$, then we compute*

$$\begin{aligned}
\text{KeyIDs}(PS_{\mathsf{P.PID}}) &= \text{KeyIDs}(P_{\mathsf{P.PID}}) \cap \text{KeyIDs}(S_{\mathsf{P.PID}}) \\
&= \{\mathtt{A},\mathtt{B},\mathtt{C}\} \cap \{\mathtt{A}\} = \{\mathtt{A}\} \\
\text{KeyIDs}(PS_{\mathsf{S.SID}}) &= \text{KeyIDs}(P_{\mathsf{S.SID}}) \cap \text{KeyIDs}(S_{\mathsf{S.SID}}) \\
&= \{\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4},\mathtt{5}\} \cap \{\mathtt{2},\mathtt{3}\} = \{\mathtt{2},\mathtt{3}\}
\end{aligned}$$

Then these KeyID lists are propagated to all entities. For example, propagating the P.PID*s* $\{\mathtt{A}\}$ *to* C.CID *results in* $\text{pKeyIDs}_1(PS_{\mathsf{C.CID}}) = \{\mathtt{1},\mathtt{2}\}$*. Propagating* S.SID*s* $\{\mathtt{2},\mathtt{3}\}$ *results in* $\text{pKeyIDs}_2(PS_{\mathsf{C.CID}}) = \{\mathtt{1}\}$*. To obtain the other KeyID lists we perform intersections. Take e.g. for* C.CID*:*

$$\begin{aligned}
\text{KeyIDs}(PS_{\mathsf{C.CID}}) &= \text{pKeyIDs}_1(PS_{\mathsf{C.CID}}) \\
&\quad \cap \text{pKeyIDs}_2(PS_{\mathsf{C.CID}}) \\
&\quad \cap \text{KeyIDs}(P_{\mathsf{C.CID}}) \cap \text{KeyIDs}(S_{\mathsf{C.CID}}) \\
&= \{\mathtt{1},\mathtt{2}\} \cap \{\mathtt{1}\} \cap \{\mathtt{1},\mathtt{2}\} \cap \{\mathtt{1}\} = \{\mathtt{1}\}
\end{aligned}$$

*Thus, the support of {(*P.Name=Jan*), (*S.Surname=A*)}*$_{\mathsf{C.CID}} = |\{\mathtt{1}\}| = 1$*. It is clear that this result corresponds to what can be seen in the full outer join in Figure 4.4.*

The time complexity of SMuRFIG is as follows. For a single itemset, at most three intersections are required for each entity E, taking $O(\sum_{E\in\mathcal{E}} |E|)$, where $|E|$ is the number of tuples in E. The propagation function, which is executed at most twice, uses each relation R exactly once, amounting to $O(\sum_{R\in\mathcal{R}} |R|)$. In order to decide whether an entity lies between two other entities, we must compute the paths between all pairs of entities in the database scheme's graph. Fortunately, this must be done only once and takes $O(|\mathcal{E}|^2)$ time, since we are considering simple schemes. To sum up, the time complexity of SMuRFIG is $O\left(|\mathcal{E}|^2 + |\mathcal{F}| \cdot size(\mathcal{DB})\right)$, where $|\mathcal{F}|$ is the total number of frequent itemsets and $size(\mathcal{DB}) = \sum_{E\in\mathcal{E}} |E| + \sum_{R\in\mathcal{R}} |R|$ is the size of the database.

SMuRFIG only requires a small amount of patterns to be stored in memory simultaneously. At the time of the generation of an itemset of length k, we have $\frac{k^2+k}{2}$ previous itemsets in memory due to the depth-first traversal. For each of these itemsets we must store KeyID lists for all keys. The maximal total size of these lists for one itemset is $\sum_{E\in\mathcal{E}} |E|$. Next to this, we also keep all relations R in memory, which are needed in the propagation function. To sum up, if l is the size of the largest frequent itemset, then SMuRFIG's *worst case* memory consumption is $O\left(l^2 \cdot \sum_{E\in\mathcal{E}} |E| + \sum_{R\in\mathcal{R}} |R|\right)$. Note, however, that in practice the size of a KeyID list can be much smaller than the corresponding entity $|E|$.

To reduce the algorithmic complexity somewhat, we can also consider a simpler version of the Keyclat algorithm. We can opt to determine the KeyID list of an itemset by only taking the intersection of the KeyID-lists of the frequent subsets. This choice, however, changes the semantics of support of a relational itemset. For example, if the support of the itemset $\{(\mathsf{C.Credits} = \mathtt{10}), (\mathsf{C.Project} = \mathtt{Y})\}_{\mathsf{P.PID}}$ is computed by the intersection of the PID-lists of $\{(\mathsf{C.Credits} = \mathtt{10})\}_{\mathsf{P.PID}}$ and $\{(\mathsf{C.Project} = \mathtt{Y})\}_{\mathsf{P.PID}}$ then it is to be interpreted as the number of professors that give a course with 10 credits and also a (possibly different) course that has a project. While in the normal, stricter semantics this should be the same course. We call this second semantics the *loose semantics*.

Definition 4.6. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$ the **absolute loose support** of a relational itemset $\{(A_1 = v_1), \ldots, (A_n = v_n)\}_K$ is the number of distinct values of the key K in the answer of the relational algebra query:*

$$\bigcap_{i=1\ldots n} \pi_K \sigma_{A_i=v_i} E_1 \bowtie_{key(E_1)} R_{1,2} \bowtie_{key(E_2)} E_2 \bowtie \cdots \bowtie E_n$$

Note that the loose support of an itemset is always greater than or equal to its strict support. Loose relative support and frequency are defined similarly to Definition 4.3. The algorithm *Keyclat-loose* (see Algorithm 4.5) computes the KeyID intersections of I for every entity key. It basically comes down to the the Eclat algorithm, running in parallel for every possible key. There is only one initial propagation, taking $\mathcal{O}\left(\sum_{E\in\mathcal{E}} |\mathcal{I}_E| \cdot \sum_{R\in\mathcal{R}} |R|\right)$, otherwise the algorithm performs an intersection for every key. Hence, computing the support of an itemset with the loose semantics is in $\mathcal{O}(\sum_{E\in\mathcal{E}} |E|)$. The whole algorithm is in $\mathcal{O}\left(\sum_{E\in\mathcal{E}} |\mathcal{I}_E| \cdot \sum_{R\in\mathcal{R}} |R| + |\mathcal{F}'| \cdot \sum_{E\in\mathcal{E}} |E|\right)$ which is less than in the strict semantics, although the number of loosely frequent itemsets $|\mathcal{F}'|$ can be much higher than the number of frequent itemsets $|\mathcal{F}|$. We investigate this impact in Section 4.5.

Since we have defined the *loose semantics* we can also consider a loose naive algorithm, given as Algorithm 4.6. Similar to the naive algorithm, the algorithm computes the full outer join. After that it will run a specialised version of Eclat for each key(E) (line 4). This Eclat version (Eclat$_K$) will count the KeyIDs for the given $K = key(E)$ instead of the row-ids of J, but is otherwise identical to Eclat as presented in Chapter 2. After that, the itemsets in key(E) are added to the output. This algorithm has the same drawbacks as the naive algorithm, while these do not occur with SMuRFIG under loose semantics.

4.2.3 NULL values

In real world datasets it is often the case that certain attributes of an entity only apply in certain cases. Some subset of the tuples will have values, while the other tuples typically have a `NULL` value. In the naive approach, the full outer join actually will generates `NULL` values. In that case we ignore `NULL` as a valid value for an attribute in an itemset. We will do the same for `NULL` values that are already

Algorithm 4.5 Keyclat-loose

Input: Set of k-itemsets $\mathcal{L}_P$ with a common prefix P; support threshold *minsup*

Output: Set $\mathcal{F}$ of all I $\in \mathcal{I}$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is loosely frequent and common prefix P

```
1:  for $I'$ in $\mathcal{L}_P$ do
2:    for $I''$ in $\mathcal{L}_P$ with $I'' > I'$ do
3:      $I := I' \cup I''$
4:      for $E \in \mathcal{E}$ do
5:        $K := key(E)$
6:        $\text{KeyIDs}(I_K) := \text{KeyIDs}(I'_K) \cap \text{KeyIDs}_K(I''_K)$
7:        $support(I_K) := |\text{KeyIDs}(I_K)|$
8:        if $support(I_K) \geq minsup \times |E|$ then
9:          $\mathcal{F}_{I'} := \mathcal{F}_{I'} \cup I_K$
10:     $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_{I'} \cup \text{Keyclat-loose}(\mathcal{F}_{I'}, minsup)$
11: return $\mathcal{F}$
```

Algorithm 4.6 Naive Loose Relational Itemset Miner

Input: An instance of a simple relational scheme $(\mathcal{E}, \mathcal{R})$; relative threshold *minsup*

Output: Set $\mathcal{F}$ of all I $\in \mathcal{I}$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is loosely frequent

```
1: $J := E_1 \bowtie R_{1,2} \bowtie E_2 \bowtie \cdots \bowtie E_n$
2: for all $E \in \mathcal{E}$ do
3:   $K := key(E)$
4:   $\mathcal{I}_K := \text{Eclat}_K(J, \text{minsup}\times|E|\ )$
5:   $\mathcal{F} := \mathcal{F} \cup \mathcal{I}_K$
6: return $\mathcal{F}$
```

present in the data. This also allows us to do regular itemset mining by creating an entity that has all possible items as attributes and the TID as key. Tuples will have a predefined value for each attributes that occurs (e.g. `1`), and `NULL` if the attribute does not.

4.3 Deviation

The relational case we are considering also brings additional challenges. For instance, let us consider the itemset $\{(\text{C.project} = \texttt{Y})\}_{\text{S.SID}}$ having a support of 67%, telling us that 67% of the students take a course with a project. When assuming a support threshold of 50%, this would be a frequent itemset. It is, however, not necessarily an *interesting* itemset. Suppose we also find that $\{(\text{C.project} = \texttt{Y})\}$ holds for 30% of the courses. Depending on the connectedness of students and courses a support of 67% could even be the *expected* value. For example, if students typically take 1 course then the expected % (if one assumes no bias) would be 30%. However, if students take 2 courses, it rises to 51.2%, with 3 courses 66.1% and so on. So in the case of an average of 3 courses per student, the 67% is expected, and thus we could consider this pattern to be uninteresting. Hence, in order to determine if $\{(\mathsf{C.project} = \mathtt{Y})\}_{\mathsf{S.SID}}$ is interesting, we use the connections and the support in $\mathsf{C.CID}$ to compute the *expected support* of $\{(\mathsf{C.project} = \mathtt{Y})\}_{\mathsf{S.SID}}$. Then, we will only consider the itemset to be *interesting* if the real support deviates substantially from the expected support.

To formalise this notion, we start off by only considering relational itemsets I consisting of items from a single entity E with key K. We refer to these kinds of itemsets as *intra-entity* itemsets. We now only want to find those frequent itemsets $I_{K'}$ where the support in K' deviates enough from the expected support in K'. We now formally define expected support in our context (based on the general definition for expected value) as follows:

Definition 4.7. *Let $I \subseteq \mathcal{I}$ be an intra-entity itemset containing items from a single relation E with key K, and let K' be the key of some other entity E'. Furthermore, let S and S' be two random variables for the support of I_K and $I_{K'}$, respectively. Then the **expected absolute support** of $I_{K'}$, given that support(I_K) $= s$, equals*

$$E[S'|S = s] = \sum_{i=1}^{k'} \left(1 - \prod_{j=0}^{d_i - 1} \left(1 - \frac{s}{k-j}\right)\right)$$

where $k = |E|$, $k' = |E'|$, and d_i is the degree *of the i-th tuple in E', i.e. the number of tuples in E that tuple i is connected to. Note that when $k - d_i < s$ then $\prod_{j=0}^{d_i-1}(1 - \frac{s}{k-j}) = 0$.*

The formula above is derived as follows. The relative support s/k is equal to the probability $P(I)$ of a tuple in E. The probability that a connection of a tuple of E' goes to a tuple of E where I does *not* hold is $(1 - s/k)$. The probability that a second connection of a tuple of E' goes to a tuple of E where I does not hold, given that there already was a first connection that did not hold is $(1 - \frac{s}{k-1})$. Since for tuple i of E' there are d_i connections to tuples of E, the probability that none of them are to a tuple where I holds is $\prod_{j=0}^{d_i-1}(1 - \frac{s}{k-j})$, and therefore the probability that one of them *does* connect is $1 - \prod_{j=0}^{d_i-1}(1 - \frac{s}{k-j})$. We then take the sum over all tuples i of E' and obtain the stated formula.

Using this definition of expected support, we formally introduce *deviation.*

Definition 4.8. *Given an itemset $I \subseteq \mathcal{I}$, let E_K be the entity of key K. We define the **deviation** of I_K as*

$$\frac{|support(I_K) - E[support(I_K)]|}{|E_K|}.$$

Our goal is to find frequent relational itemsets with a defined minimal deviation, in order to eliminate uninteresting relational itemsets. Results of experiments performed in this setting are reported in Section 4.5.

Note that we restricted ourselves to a special case of itemsets only containing items with attributes from the same entity (*intra-entity* itemsets). In order to generalise deviation and expected support to itemsets containing attributes from multiple entities, which we refer to as *inter-entity* itemsets, it is necessary to generalise the definition of support of itemsets to allow sets of keys.

Definition 4.9. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, and a set of keys $\mathcal{K} \subseteq \bigcup_{E \in \mathcal{E}}\{\{key(E)\}\}$, the* ***support*** *of a relational itemset $I_{\mathcal{K}} = \{(A_1 = v_1), \ldots, (A_n = v_n)\}_{\mathcal{K}}$ is the number of distinct values of $\mathcal{K}$ in the answer of the query:*

$$\pi_{\mathcal{K}}\sigma_{A_1=v_1,\ldots,A_n=v_n} E_1 \bowtie_{key(E_1)} R_{1,2} \bowtie_{key(E_2)} E_2 \bowtie \cdots \bowtie E_n$$

where $E_i \bowtie_{key(E_i)} R_{i,i+1}$ represents the equi-join on $E_i.\mathrm{key}(E_i) = R_{i,j}.\mathrm{key}(E_i)$ and all $E_i \in \mathcal{E}_{I_{\mathcal{K}}}$ are joined using the unique path $P_{I_{\mathcal{K}}}$.

The ***relative support*** *of an itemset is the absolute support of the itemset divided by the number of distinct values for $\mathcal{K}$ in the answer of the query:*

$$\pi_{\mathcal{K}} R_{1,2} \bowtie_{key(E_2)} R_{2,3} \bowtie \cdots \bowtie R_n$$

Using this definition we can define expected support for inter-entity itemsets:

Proposition 4.2. *Let $I \subseteq \mathcal{I}$ be an itemset consisting of items with attributes belonging to the set of entities $\mathcal{E}_{\mathcal{I}} = \{E_1, \ldots, E_n\}$, and let E be an entity with key K. Furthermore, let $keys(I)$ denote $\bigcup_{E_i \in \mathcal{E}_{\mathcal{I}}} key(E_i)$, then let S and S' be two random variables for the support of $I_{\mathcal{K}}$ and I_K respectively, where $\mathcal{K} = keys(I)$. Then the expected absolute support of $I_{\mathcal{K}}$, given that the absolute support of $I_{\mathcal{K}} = s$ $(s \in [0, k_1])$ and given the connections in the join table connecting the entities, is given as:*

$$E[S'|S = s] = \sum_{i=1}^{k'} \left(1 - \prod_{j=0}^{d_i - 1}\left(1 - \frac{s}{k-j}\right)\right)$$

where k = number of possible values for $\mathcal{K}$ in $R_{1,2} \bowtie_{key(E_2)} R_{2,3} \bowtie \cdots \bowtie R_n$, and k' is the number of tuples in E and d_i is the degree *of the ith tuple in E, i.e. the number of tuples in $E_1 \bowtie_{key(E_1)} R_{1,2} \bowtie_{key(E_2)} E_2 \bowtie \cdots \bowtie E_n$ this tuple is connected to.*

This is essentially the same formula as Proposition 4.7, only now using the support in a set of keys. Thus, the computation of expected support remains roughly the same. Nevertheless, in order to execute this computation for an itemset I, we need to obtain the support of I in the set of keys belonging to the entities of the attributes it contains (*keys*(I)). One option is to adapt our algorithm such that it considers the support of an itemset in every possible set of keys. This option creates too much overhead, since we do not need all these supports for all itemsets. To be able to compute expected support of an itemset I we only additionally need the support in *keys*(I). Therefore we consider a second option, where we adapt our propagation procedure such that it also additionally computes the support in *keys*(I) for every inter-entity itemset I.

So far, we assumed that KeyID lists did not contain any duplicates. Yet, when translating from one KeyID list to another, duplicates arise naturally.

Example 4.8. *Considering the instance from Figure 4.3, suppose we are translating the Course.CID list (1,2,3) to Professor.PID. Since both course 1 and 2 are connected to professor A, and both course 2 and 3 are connected to professor B, translating this without eliminating duplicates would result in the Professor.PID list (A,A,B,B).*

If we assume that KeyID lists contain duplicates, we can use these KeyID lists to compute the support in *keys*(I) of an inter-entity itemset I in order to compute its expected support.

Example 4.9. *Consider the itemset {(P.Name=Jan), (S.Surname=A)}. The support in the set of keys (P.PID,S.SID) can be computed based on the KeyID list ((A,2),(A,3)) and thus equals 2, as can be verified in the full outer join of Figure 4.4. When not eliminating duplicates the Professor.PID list of this itemset would be (A,A), and the Student.SID list would be (2,3). It is clear that these KeyID lists are the decomposition of the KeyID list in (P.PID,S.SID). Thus we can compute the support in (P.PID,S.SID) as the number of elements in the KeyID of P.PID or S.SID, taking duplicates into account. On the other hand, the support in e.g. P.PID would be the number of unique elements in the P.PID list.*

Considering duplicates in KeyID lists, then

$$support(I_{keys(I)}) = |\mathrm{KeyID}_K(I)| \text{ where } K \in keys(I) \tag{4.1}$$

Let uniq(L) be a function that eliminates duplicates in a list L, then for all $E \in \mathcal{E}$

$$support(I_{key(E)}) = |\mathrm{uniq}(\mathrm{KeyID}_E(I))| \tag{4.2}$$

In order to use this new computation in our algorithm, we need to adapt it such that it generates KeyID lists with duplicates. However, really storing duplicates in the KeyID lists would result in bad memory and computation requirements. In the worst case, a KeyID list would become as large as the number of tuples in the full outer join of all relations, which we denote by $size(⟗_{\mathcal{DB}})$. Instead of really working with duplicated elements, we will consider KeyID lists where each element has an associated number of occurrences. The list (A,A,B,B) from Example 4.8, would be stored as (A,B) with the associated occurrences list (2,2). This storage structure ensures that the memory requirement for the largest KeyID list is only $|E_{max}| \cdot \log(size(\bowtie_{\mathcal{DB}}))$, where E_{max} is the entity containing the largest number of tuples.

To adapt the SMuRFIG algorithm, we first change the translate function such that it does not eliminate duplicates when translating KeyIDs, and instead stores the needed number of occurrences for each key.

Example 4.10. *Considering the course list Course.CID list (1,2,3) with occurrences (1,2,2). The translation to Professor.PID will occur in SMuRFIG, but the occurences of the translation of course 2 and 3 is multiplied by 2. This results in the Professor.PID list (A,B) with occurrences (2,5).*

Because we need to take these occurrences into account, the Propagate function needs to be adapted to use previously computed connections, otherwise unwanted duplicates can arise.

Example 4.11. *Consider the itemset {(P.Name=Jan), (S.Surname=LP)}. In the set of keys (P.PID,S.SID) the KeyID list is ((A,1),(B,1)). If one computed the support pairwise, while not eliminating duplicates the Professor.PID list of this itemset would be (A,A,B), and the Student.SID list would be (1,1,1). These KeyID lists are decomposition of the KeyID list ((A,1),(A,1),(B,1)) which clearly contains an unwanted duplicate.*

This entails that we perform the pairwise propagation once for every key-pair, *i.e.*, we compute which tuples in entity E_i are connected to which tuples in entity E_j. These connections can essentially be computed using the original Propagate algorithm run using the full KeyID list of an entity as input. This initial computation has a complexity of $O(\sum_{E \in \mathcal{E}} |E| \cdot \sum_{R \in \mathcal{R}} |R|)$. Later we use these computed connections to perform the KeyID list propagations for distinct itemsets.

Most changes are required in the Keyclat algorithm, where we intersect KeyID lists. In the updated algorithm, given as Algorithm 4.7, most of the intersections remain unchanged and ignore the occurrence numbers. This is the case for the intersection on line 7 and on line 20. The intersection, originally on line 12 in Algorithm 4.3, is now changed into a small join, shown on lines 12 to 15. This ensures that each element k in $\text{pKeyIDs}_E(I')_1 \cap \text{pKeyIDs}_E(I')_2$ appears $o_1(k) \times o_2(k)$ times where $o_i(k)$ is the number of occurrences of k in $\text{pKeyIDs}_E(I')_i$. On line 18 we use the occurrences of the $\text{pKeyIDs}_E(I')_i$ containing the most duplicates, *i.e.*, where $\sum_{k \in \text{pKeyIDs}_E(I')_i} o_i(k)$ is the largest. The original intersection is changed such that the number of occurrences are preserved in the propagated KeyID lists (lines 19 to 22). Finally in order to determine the support values only the KeyID list is used and the occurrences are ignored (cfr. Equation 4.2).

This modified algorithm ensures that we can compute support as before by only looking at the KeyID lists

$$support(I_{key(E)}) = |\mathrm{KeyID}_E(I)| \tag{4.3}$$

and that we can, additionally, by looking at the occurrence lists, know the number of duplicates in the KeyID lists. Let $o_K(k)$ denote the number of occurrences of k in $\mathrm{KeyIDs}_K(I)$. Then the support of an itemset I in the keys of the itemset's entities ($keys(I)$) can be computed as follows:

$$support(I_{keys(I)}) = \sum_{k \in \mathrm{KeyIDs}_K(I)} o_K(k) \text{ where } K \in keys(I) \tag{4.4}$$

This provides the component s_1 to be able to compute the expected support of an itemset. We can compute the component k_1 by considering the KeyID list of all tuples of E_1 and then propagating to E_n. The support of the resulting $KeyID$ list considering occurrences represents k_1. Propagating this KeyID list to E we can derive the d_i by counting the amount of duplicates of each tuple i of E in this result.

Given all these components, we are now able to compute the expected support of any itemset, and to perform a minimal deviation check as shown on line 30.

The storage of occurrence lists changes the complexity of our algorithm. KeyID lists can now become larger in size than the number of tuples in the entities. As stated the size of largest occurrence list is $|E_{max}| \cdot \log(size(\bowtie_{\mathcal{DB}}))$. For each entity E separately this is $|E| \cdot \log(size(\bowtie_{\mathcal{DB}}))$. Storing the direct links requires $O(|E_{max}| \cdot \sum_{R \in \mathcal{R}} |R|)$, where $|E_{max}|$ is the size of the largest entity. This results in the following memory requirements for SMuRFIG with occurrence lists:

$$\mathcal{O}\left((\frac{n^2}{2} + \frac{n}{2}) \cdot (\sum_{E \in \mathcal{E}} |E|) \cdot \log(size(\bowtie_{\mathcal{DB}})) + |E_{max}| \cdot \sum_{R \in \mathcal{R}} |R| \right)$$

The multiplications of occurrences take $O(\log(size(\bowtie_{\mathcal{DB}}))^2)$ which result in $O(\log(size(\bowtie_{\mathcal{DB}}))^2) \cdot \sum_{E \in \mathcal{E}} |E|)$ time complexity for the intersections. The propagation of KeyID lists now taking the occurrences into account takes $O(|E_{max}| \cdot \log(size(\bowtie_{\mathcal{DB}})) \cdot \sum_{R \in \mathcal{R}} |R|)$. Together with the $O(\sum_{E \in \mathcal{E}} |E| \cdot \sum_{R \in \mathcal{R}} |R|)$ required to compute the direct links, this results in the worst-case algorithmic complexity of

Algorithm 4.7 Keyclat Deviation

Input: Set of k-itemsets $\mathcal{L}_P$ with a common prefix P; support threshold *minsup*; deviation threshold *mindev*

Output: Set $\mathcal{F}_\mathcal{D}$ of all itemsets I $\in \mathcal{I}$ where $\exists E \in \mathcal{E} : I_{key(E)}$ is frequent and deviating and has prefix P

```
1:  for $I'$ in $\mathcal{L}$ do
2:    for $I''$ in $\mathcal{L}$ with $I'' > I'$ do
3:      $I := I' \cup I''$
4:      $E_1, E_2$ := entity of suffix items $I' \setminus P, I'' \setminus P$ respectively
5:      for $i \in \{1, 2\}$ do
6:        $K_i := key(E_i)$
7:        pKeyIDs$(I)_i$ := Propagate(KeyIDs$(I'_{K_i})$ $\cap$ KeyIDs$(I''_{K_i})$)
8:      for $E \in \mathcal{E}$ do
9:        $K := key(E)$
10:       if $E$ lies between $E_1$ and $E_2$ then
11:         pKeyIDs$(I_K)$ := (pKeyIDs$(I_K)_1$ $\cap$ pKeyIDs$(I_K)_2$)
12:         for all $k \in$ pKeyIDs$(I_K)$ do
13:           $o_1$ := number of occurrences of $k$ in pKeyIDs$(I_K)_1$.
14:           $o_2$ := number of occurrences of $k$ in pKeyIDs$(I_K)_2$.
15:           number of occurrences of $k$ in pKeyIDs$(I_K)$ := $o_1 \times o_2$
16:       else
17:         pKeyIDs$(I_K)_i$ is the largest
18:         pKeyIDs$(I_K)$ := pKeyIDs$(I_K)_i$
19:       for all $k \in$ pKeyIDs$(I_K)$ do
20:         if $k \in$ (KeyIDs$(I'_K)$ $\cap$ KeyIDs$(I''_K)$) then
21:           KeyIDs$(I_K)$ := KeyIDs$(I_K)$ $\cup \{k\}$
22:           number of occurrences of $k$ in KeyIDs$(I_K)$ :=
                  number of occurrences of $k$ in pKeyIDs$(I_K)$
23:       $support(I_K)$ := |KeyIDs$(I_K)$|
24:       if $support(I_K) \geq minsup \times |E|$ then
25:         $\mathcal{F}_{I'} := \mathcal{F}_{I'} \cup I_K$
26:   $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_{I'} \cup$ KeyclatDeviation($\mathcal{F}_{I'}$, $minsup$, $mindev$)
27: for all $I \in \mathcal{F}$ do
28:   for $E \in \mathcal{E}$ do
29:     $K := key(E)$
30:     if $|support(I_K) -$ ExpectedSupport$(I)| \geq (mindev \times |E|)$ then
31:       $\mathcal{F}_\mathcal{D} := \mathcal{F}_\mathcal{D} \cup I_K$
32: return $\mathcal{F}_\mathcal{D}$
```

SMuRFIG with occurrence lists is

$$\mathcal{O}\Bigg(|\mathcal{E}|^2 + |\mathcal{F}| \cdot \Bigg(\log(size(\bowtie_{\mathcal{DB}}))^2 \cdot \sum_{E\in\mathcal{E}} |E| + |E_{max}| \cdot \log(size(\bowtie_{\mathcal{DB}})) \cdot \sum_{R\in\mathcal{R}} |R|\Bigg) + \sum_{E\in\mathcal{E}} |E| \cdot \sum_{R\in\mathcal{R}} |R|\Bigg)$$

4.3.1 Rule Deviation

For use in their Warmr algorithm, [Dehaspe & Toivonen, 2001] consider deviation on rules. Consider the rule $A \Rightarrow_K C$, where $K = key(E)$. The expected value of $support(AC)$ can be considered to be calculated as the support of A times the probability of C holding. The probability of C holding is expressed as fraction of E for which C holds, or $support(C)/|E|$. Supposing S_1, S_2, S_3 random variables for respectively the support values of AC, A, C, the formula becomes

$$E[S_1|S_2 = support(A), S_3 = support(C)] = support(A)\frac{support(C)}{|E|}$$

Then we can define the *deviation* as:

$$support(AC) - E[support(AC)] = support(AC) - support(A)\frac{support(C)}{|E|}$$

We note that this formula is a variant of the *leverage* measure [Piatetsky-Shapiro, 1991]. As we define a generalisation of leverage, called minimal divergence, in Section 4.4.2 we will not be considering rule deviation, instead only considering a minimal divergence threshold.

4.4 Redundancy

Examining the initial results of our algorithm, we uncovered several types of redundancies. In our multi-relational setting, the type of redundancies we already encounter in the standard itemset and association rule mining setting are also present and worse due to the relational multiplication factor (the different keys and the different relations).

4.4.1 Closure

We extend the notion of closed itemsets (see Chapter 2) to our multi-relational setting and apply it to prune itemsets and generate association rules. We say that an itemset I is **closed** if for all supersets I' of I there exists a key K such that $support(I'_K) < support(I_K)$.

Definition 4.10. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, a relational itemset $\{(A_1 = v_1), \ldots, (A_n = v_n)\}$ is* ***closed*** *if for all $(A_m = v_m)$ there exists a key $K \in \bigcup_{E \in \mathcal{E}} \{key(E)\}$ such that*

$$\begin{aligned} &support(\{(A_1 = v_1), \ldots, (A_n = v_n), (A_m = v_m)\}_K) \\ &\qquad\qquad\qquad < support(\{(A_1 = v_1), \ldots, (A_n = v_n)\}_K). \end{aligned}$$

The **closure** of an itemset is defined as its smallest closed superset. It is noteworthy that while a non-closed itemset produces an association rule with confidence 100%, the opposite is not necessarily true. For instance, given the itemsets A, C and a key K, it is perfectly possible that the rule $A \Rightarrow_K C$ has a confidence of 100%, without C being in the closure of A.

Example 4.12. *In the context of the instance from Figure 4.3, $\{(\mathsf{P.Name} = \mathsf{Jan})\} \Rightarrow_{\mathsf{P.PID}} \{(\mathsf{C.Credits} = \mathsf{10})\}$ is a 100% rule, since the professors with name Jan (A, B and C) are the same as the professors with name Jan that also give a course with 10 credits. However, $\{(\mathsf{C.Credits} = \mathsf{10})\}$ is not in the closure of $\{(\mathsf{P.Name} = \mathsf{Jan})\}$ since the rule $\{(\mathsf{P.Name} = \mathsf{Jan})\} \Rightarrow_{\mathsf{C.CID}} \{(\mathsf{C.Credits} = \mathsf{10})\}$ only has 75% confidence, since there are 4 courses (1,2,3,4) taught by a 'Jan', but only 3 courses (1,2,4) taught by a 'Jan' that have 10 credits.*

Essentially C is only in the closure of A if for all keys K, $A \Rightarrow_K C$ is a 100% association rule. In the case that only for a specific key K, $A \Rightarrow_K C$ is 100%, then for an itemset C', the confidence of $A \Rightarrow_K CC'$ is not necessarily equal to the confidence of $A \Rightarrow_K C'$.

Example 4.13. $\{(\mathsf{P.Name} = \mathsf{Jan})\} \Rightarrow_{\mathsf{P.PID}} \{(\mathsf{S.Surname} = \mathsf{VG})\}$ *is a 33% rule, while* $\{(\mathsf{P.Name} = \mathsf{Jan})\} \Rightarrow_{\mathsf{P.PID}} \{(\mathsf{C.Credits} = \mathsf{10}), (\mathsf{S.Surname} = \mathsf{VG})\}$ *does not hold.*

From Definition 4.10 it follows that, if C is in the closure of A, then it does hold that the confidence (and support) of $A \Rightarrow CC'$ is the same as that of $A \Rightarrow C'$, and hence the former association rule is redundant. In our algorithm, we will employ the same closure-based redundancy techniques as proposed by Zaki (see Chapter 2), keeping in mind, however, that not all 100% rules entail a redundancy. Specifically, a rule $A \Rightarrow C$ is redundant if A is not a generator, or if C contains an item from the closure of A (unless $|C| = 1$). Additional to closure redundancy, we do not generate rules of the form $key(R) \Rightarrow A$ where $A \subseteq sort(R)$, since these rules are self evident. We also include the option to completely exclude keys from the itemsets/rules. This allows for more focus on the relations between attributes, but the results are of course incomplete.

4.4.2 Divergence

Next to these lossless redundancy removal techniques, we also introduce an extra, but lossy, pruning technique. The technique is a generalisation of minimal improvement [Bayardo Jr et al., 2000]. In minimal improvement, all rules must have a confidence that is at least a minimal improvement threshold value greater than any of its proper subrules. A proper subrule of a rule $A \Rightarrow C$ is defined as any rule $B \Rightarrow C$ where $B \subset A$. The aim of this measure is to eliminate rules ($A \Rightarrow C$) that have additional constraints than a simpler rule ($B \Rightarrow C$), but do not differ much in confidence. Minimal improvement, as the name implies, only allows for rules to improve in confidence. While this makes sense in a market-basket analysis setting, a 'negative improvement' can also be interesting. Suppose the rule $\{(\textsf{Stdy.YID}{=}\texttt{I})\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits}{=}\texttt{10})\}$ has a confidence of 40%, telling us that 40% of the students of study I take a course with 10 credits. Suppose now we also have the rule $\{\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits}{=}\texttt{10})\}$ which has 90% confidence. This would imply that the students of the study I significantly diverge from the general population of all students, which could be an interesting fact that could be subject for further study. But it is a negative divergence, and therefore the rule $\{(\textsf{Stdy.YID}{=}\texttt{I})\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits}{=}\texttt{10})\}$ would not be generated when considering minimal improvement, since this rule does not improve upon the simpler rule $\{\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits}{=}\texttt{10})\}$. To also allow significant negative divergences we define *minimal divergence* as follows: all rules $A \Rightarrow C$ must have a confidence c such that for all proper subrules with confidence c' it holds that $|c - c'|$ is greater than a given minimal divergence threshold. This pruning measure has the effect of eliminating rules that have additional constraints but a similar confidence to a simpler rule, which we prefer. For instance, instead of several rules $\{(\textsf{Stdy.YID} = \texttt{I})\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits} = \texttt{10})\}$, $\{(\textsf{Stdy.YID} = \texttt{II})\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits} = \texttt{10})\}$, $\{(\textsf{Stdy.YID} = \texttt{III})\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits} = \texttt{10})\}$,... that all have very similar confidence we now only have one rule $\{\} \Rightarrow_{\textsf{S.SID}} \{(\textsf{C.Credits} = \texttt{10})\}$ with the weighted average as the confidence.

We note, that divergence can be seen as a generalisation of the *leverage* measure [Piatetsky-Shapiro, 1991]. For a rule $A \Rightarrow_K B$, where we consider E to be the entity of the key K in which the rule is expressed, leverage is defined as:

$$\mathit{leverage}(A \Rightarrow_K B) = \frac{\mathit{support}(AB)}{|E|} - \frac{\mathit{support}(A)}{|E|}\frac{\mathit{support}(B)}{|E|}$$

If we divide this by $\frac{\mathit{support}(A)}{|E|}$ we obtain

$$\frac{\mathit{support}(AB)}{\mathit{support}(A)} - \frac{\mathit{support}(B)}{|E|} = \mathit{conf}(A \Rightarrow B) - \mathit{conf}(\{\} \Rightarrow B)$$

The generalisation of divergence thus lies in the fact that we not only compare $A \Rightarrow B$ with $\{\} \Rightarrow B$ but with any $A' \Rightarrow B$ where $A' \subset A$. More comparisons entail that a rule can have sufficient leverage, but can still be redundant when considering minimal divergence. Furthermore, minimal divergence is also related to the rule deviation discussed in Section 4.3.1. In fact, given a minimal divergence threshold of *mindiv*, the absolute value of this rule deviation is always greater than $\mathit{mindiv}/\mathit{support}(A)$. As stated before, we therefore only consider the minimal divergence threshold.

To analyse the impact of a minimal divergence threshold, we performed experiments using varying levels of this threshold and discuss the results in Section 4.5.

4.5 Experiments

In this section we consider the results of several experiments performed on real world databases using the loose and strict algorithm both implemented in C++[1], and run on a standard computer with 2GB RAM and a 2.16 GHz processor.

First, we consider a snapshot of the student database from the Mathematics and Computer Science department of the University of Antwerp. The scheme roughly corresponds to the one given in Figure 4.2. There are 174 courses, 154 students and 40 professors, 195 links between professors and courses and 2949 links between students and courses. The second database comes from the KDDcup 2003[2], and is comprised of a large collection of High Energy Physics (HEP) papers. It consists of HEP papers linked to authors and journals, and also to other papers (citations). It contains of 2543 papers, 23621 authors and 76 journals, and there are 5012 connections between authors and papers, plus 458 connections from papers to journals. The characteristics of both databases are summarised in Figure 4.5.

4.5.1 Patterns

Several interesting patterns were discovered in the Student database (using the strict semantics), and we now discuss a few of them. The algorithm returned the pattern $\{(\mathsf{C.Room} = \texttt{G010})\}_{\mathsf{S.SID}}$ with a support of 81% representing the fact that 81% of the students take a course given in room `G010`. It also discovered $\{(\mathsf{C.Room} = \texttt{G010}), (\mathsf{P.ID} = \texttt{DOE})\}_{\mathsf{S.SID}}$ with 76% support. Together they form the association rule: $\{(\mathsf{C.Room} = \texttt{G010})\} \Rightarrow_{\mathsf{S.SID}} \{(\mathsf{P.ID} = \texttt{DOE})\}$ with 94% confidence from which we conclude that in 94% of the cases, students that take a course in G010 also take a course from professor ‘`DOE`’ in `G010`. We found $\{(\mathsf{S.Study} = \texttt{1-MINF-DB})\}_{\mathsf{P.PID}}$ with a support of 63%, stating that 63% of the professors teach

[1]The source code for SMuRFIG can be downloaded at `http://www.adrem.ua.ac.be/`

[2]`http://kdl.cs.umass.edu/data/`

attribute	#values
student.*	154
student.id	154
student.name	154
student.study	6
student.contracttype	2
course.*	174
course.id	174
course.name	164
course.room	26
professor.*	40
professor.id	40
takes.*	2949
takes.studentid	154
takes.courseid	146
teaches.*	75
teaches.courseid	68
teaches.professorid	40

(a) Student DB

attribute	#values
paper.*	2543
paper.title	2542
paper.class	227
paper.published	2
paper.authors	7
paper.cited	89
paper.citing	70
paper.earliest_year	14
author.*	23621
author.id	23621
author.firstname	5085
author.lastname	7512
journal.*	76
journal.id	76
journal.name	76
journal.country	23
journal.languages	12
wrote.*	5012
wrote.authorid	2105
wrote.paperid	2543
injournal.*	458
injournal.paperid	458
injournal.journalid	36

(b) HEP DB

Figure 4.5: Number of tuples per attribute in the Student and HEP databases

a course that is taken by students of `1-MINF-DB`. We also come across this pattern with a different key $\{(\mathsf{S.Study} = \texttt{1-MINF-DB})\}_{\mathsf{S.SID}}$ with a support of 7%, telling us that only 7% of the students belong to `1-MINF-DB`. So this is a clear example of the merit of the key based frequency. Another example is the itemset $\{(\mathsf{S.Study} = \texttt{BINF})\}$ which has 68% PID support, 75% SID support and 39% CID support. Hence, 68% of all professors teach a course taken by students from `BINF`, 75% of the students are in `BINF` and 39% of the courses are taken by students from `BINF`.

Some patterns found in the HEP database include the following association rules: $\{(\mathsf{paper.earliest\_year}{=}2002)\} \Rightarrow_{\mathsf{paper.id}} \{(\mathsf{paper.published}{=}\texttt{false})\}$ and also $\{(\mathsf{paper.earliest\_year} = 2003) \Rightarrow_{\mathsf{paper.id}} (\mathsf{paper.published}{=}\texttt{false})\}$ with respectively 60% and 93% confidence. Since the dataset is from the 2003 KDD cup, this tells us that most recently submitted papers ($\mathsf{paper.earliest\_year} > 2002$) are not published yet. A rule like $\{(\mathsf{paper.authors}{=}2)\} \Rightarrow_{\mathsf{paper.id}} \{(\mathsf{paper.published}{=}\texttt{true})\}$ with 67% confidence versus the rule $\{(\mathsf{paper.authors}{=}4)\} \Rightarrow_{\mathsf{paper.id}} \{(\mathsf{paper.published}{=}\texttt{true})\}$ with 80% confidence show us that the acceptance probability of papers with 4 authors is higher. The rule $\{(\mathsf{paper.cited}{=}0)\}{\Rightarrow_{\mathsf{paper.id}}} \{(\mathsf{paper.published}{=}\texttt{false})\}$ with 60% confidence, tells us that if the paper is not cited by another paper in the database, in 60% of the cases it is not published. Furthermore, the rule $\{(\mathsf{paper.cited}{=}2)\} \Rightarrow_{\mathsf{paper.id}} \{(\mathsf{paper.published}{=}\texttt{true})\}$ with confidence 65% says that if a paper is cited by two other papers, then in 65% of the cases it is published. Further rules show us this number steadily increases and a paper cited 8 times is published in 85% of the cases. Similarly, we found rules expressing that the number of papers cited in a paper also increases its publication probability. We also found the rule $\{(\mathsf{paper.class}{=}\texttt{Quantum Algebra})\} \Rightarrow_{\mathsf{author.id}} \{(\mathsf{paper.published}{=}\texttt{true})\}$ with 75% confidence, expressing that 75% of the authors who write a paper in `Quantum Algebra` get it published.

These examples clearly demonstrate that relational itemsets and rules with key-based frequency allow us to find easy to interpret interesting patterns.

4.5.2 Interestingness

In order to relate our frequency measure to that of typical relational itemset mining techniques, we ran some experiments where we performed the full outer join of the relations and entities and then used an Eclat implementation adapted for categorical itemsets to mine frequent itemsets with frequency being expressed in number of rows [Calders et al., 2007]. The results for both databases are shown in Figure 4.6 labelled 'join'. Because of the large size of the full outer join table, the support values that result in frequent itemsets are very low, and an exponential behaviour for the lowest support thresholds can be observed. Both behaviours are due to the large number of unique tuples in the result of the full outer join. We observed that the join computation time has the largest impact factor and that it is independent of the minimal support. The number of relations and tuples in the relations determine the time as well as the space requirements. Computing the full outer join was possible for the databases examined, but for very large databases, explicitly computing this join would become impractical. Furthermore, as mentioned in the introduction, using this frequency measure, the patterns that are found, are very hard to interpret. For example we found the pattern $\{(\mathsf{author.firstname} = \mathtt{A.})\}$ with support 0.04 in the HEP database. Apart from the very low support, we also can not interpret the semantics of this pattern. Is it frequent because of a lot of authors have this first name, or because authors with this first name write a lot of papers? This problem is not present in our KeyID based approach.

To evaluate the effectiveness of deviation as an additional interestingness measure, we performed experiments on the Student and HEP databases using varying support and varying deviation thresholds. As is apparent in Figure 4.6e, increasing the minimal deviation effectively reduces the number of itemsets for the Student DB. This reduction works as promised, for instance the previously mentioned frequent itemset $\{(\mathsf{C.Room} = \mathtt{G010})\}_{\mathsf{S.SID}}$ with support 81% has a deviation of 22%, and thus could be considered interesting. The itemset $\{(\mathsf{C.Room} = \mathtt{G006})\}_{\mathsf{S.SID}}$ has 53% support, but it only has a deviation of 2%, thus is considered less interesting. As can be seen in Figure 4.6f, the impact for the HEP database seems less significant. However, this is due to the specifics of the database. A large number of the patterns found in the HEP database are ‘regular’ itemsets, *i.e.*, itemsets consisting of items from a single entity counted in the key of that entity. On these type of itemsets the deviation measure has no impact. Deviation can only impact the smaller amount of itemsets that are the result of the relational aspects. For these itemsets deviation does its job and reduces the amount of less interesting itemsets. For example the itemset $\{(\mathsf{papers.published} = \mathtt{true}), (\mathsf{papers.authors} = 2)\}_{\mathsf{journal.id}}$ has a confidence of 32% and a deviation of 11%, thus we consider it to be interesting. In contrast, the itemset $\{(\mathsf{papers.published} = \mathtt{true}), (\mathsf{papers.authors} = 1)\}_{\mathsf{journal.id}}$ with confidence 25% has a deviation of 7%, and is considered to be less interesting.

4.5.3 Redundancy

We also ran experiments on the databases to evaluate closure and minimal divergence. The results are shown in Figure 4.7. We observe that lossless closure-based redundancy pruning reduces the output by an order of magnitude, and thus works as promised. Using (only) minimal divergence we significantly reduce the rule output of our algorithm by several orders of magnitude. When examining the results we observed that the divergence based pruning indeed eliminates undesired results. For example we found the rule $\{\} \Rightarrow_{\mathsf{S.SID}} \{(\mathsf{S.contracttype}{=}\mathtt{diploma})\}$ with a confidence of 99%. Using this fact and a minimal divergence threshold of 10%

we now prune many rules like $\{(\mathsf{P.ID}{=}\mathtt{DOE})\} \Rightarrow_{\mathsf{S.SID}} \{(\mathsf{S.contracttype}{=}\mathtt{diploma})\}$ but with different P.PID's, rules like $\{(\mathsf{C.Room}{=}\mathtt{US103})\} \Rightarrow_{\mathsf{S.SID}} \{(\mathsf{S.contracttype} = \mathtt{diploma})\}$ with different rooms and so on.

4.5.4 Performance

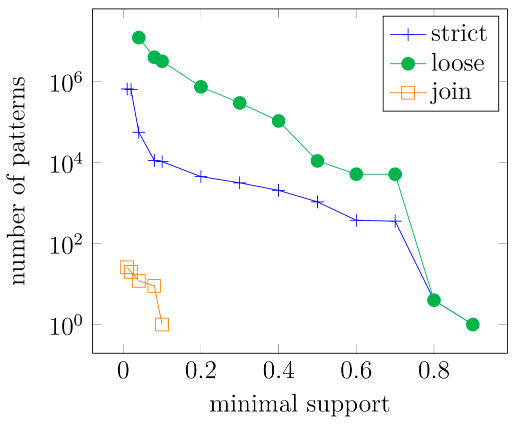
We experimentally compared the Naive and SMuRFIG algorithms on the Student and HEP datasets, gradually varying the minimum support threshold. In our experiments we also ran the standard Eclat algorithm on the full join table, an approach which has been taken in previous work. The number of patterns and the runtimes are reported below.

In Figures 4.6a and 4.6b we see that the Eclat algorithm applied directly to the join table finds far fewer patterns than SMuRFIG (and Naive which mines the same patterns) on both the Student and HEP databases. Since the (relative) minimum support threshold is set against the size of the full outer join of the database and not the size of an individual table, an itemset must be very connected to have a high support in this join. Clearly, many interesting patterns that are not highly connected will be discarded this way. Apart from this, the support of an itemset in this join table is of course less interpretable.

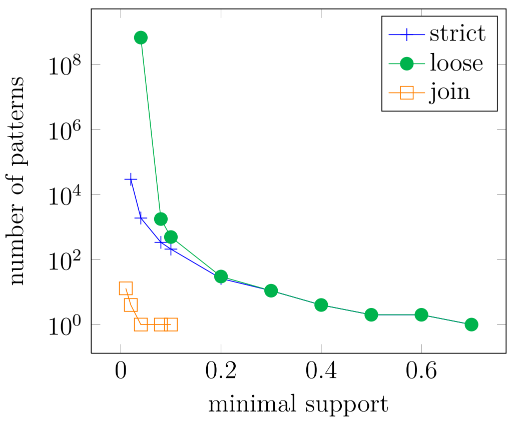
The runtimes reported in Figures 4.6c and 4.6d clearly indicate that the Naive algorithm takes a lot more time than SMuRFIG, often up to an order of magnitude or more. Although SMuRFIG performs more operations per itemset than Naive does, the Naive algorithm operates on an extremely large database compared to SMuRFIG, which pays off in terms of runtime. Note that for the HEP database, SMuRFIG is even faster that the Eclat algorithm on the join table, even though the latter finds far fewer patterns.

SMuRFIG was run using both strict and loose semantics. The difference in number of generated patterns for the Student database is presented in Figure 4.6a. As expected, the loose semantics clearly generates more patterns. For low values of minimal support, the number of patterns is even an order of magnitude larger. The loose semantics connects many more itemsets with one another than the strict semantics, resulting in even more connections for lower minimal support values. If we look at the difference in timing in Figure 4.6c, where we also include the timing of the naive algorithm, we observe that our algorithm in the strict semantics is always faster than the naive semantics. For higher values of support we also observe the loose semantics to be faster. However, for low levels of support we can see a dramatic increase in execution time for the loose semantics, surpassing that of the strict semantics. This is due to the large number of patterns the loose semantics requires to handle in the case of low support values.

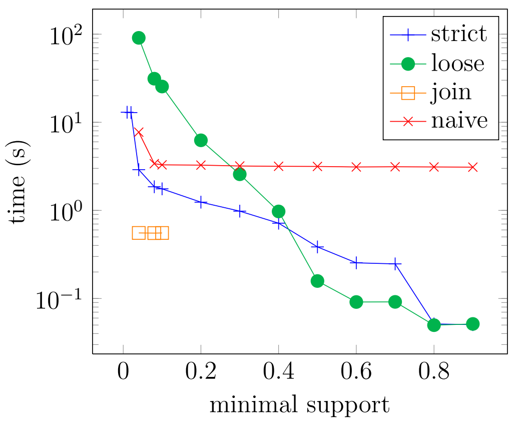
For the HEP database, we notice that the increase in the number of patterns for decreasing support pictured in Figure 4.6b is stronger as compared to the Student



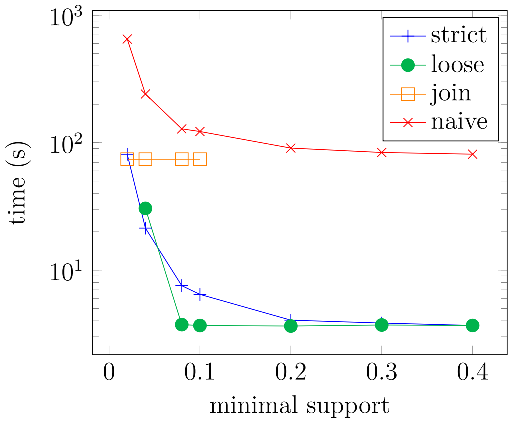
(a) Student DB: Number of patterns



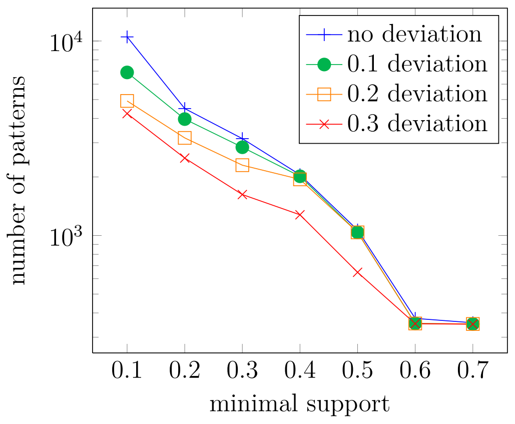
(b) HEP DB: Number of patterns



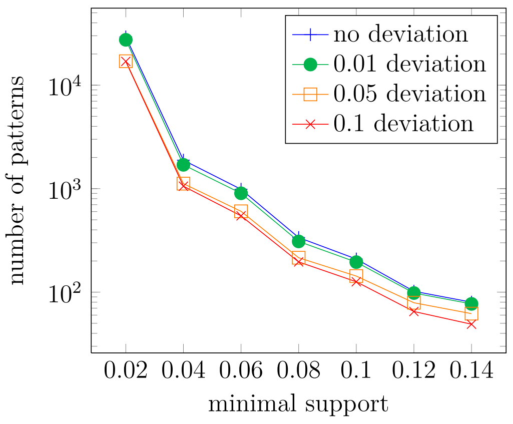
(c) Student DB: Runtime



(d) HEP DB: Runtime

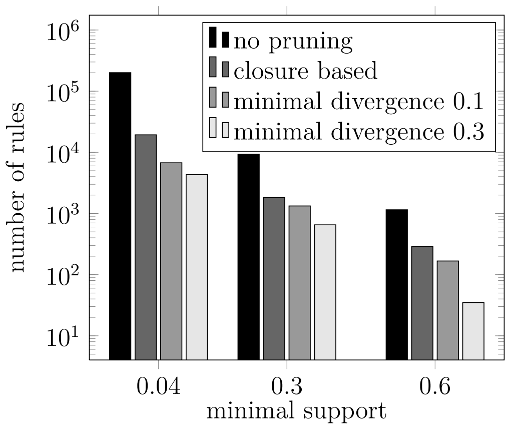


(e) Student DB: Deviation pruning

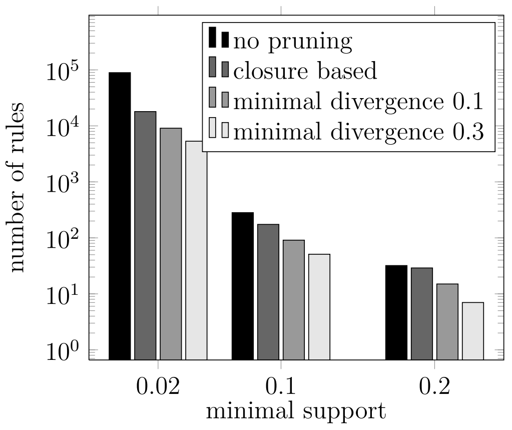


(f) HEP DB: Deviation pruning

Figure 4.6: Results for increasing minimal support



(a) Student DB: Pruning rules having a minimal confidence of 0.5



(b) HEP DB: Pruning rules having a minimal confidence of 0.4

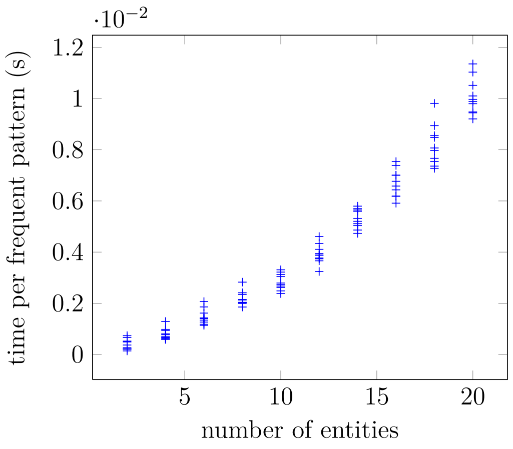
Figure 4.7: Results of pruning rules

database. In fact the increase is so drastic that we could not run the experiment using loose semantics for a minimum support of 1%. If we look at the time in Figure 4.6d, we see that for the higher support values both the strict and loose semantics are well under those of the naive algorithm. For low minimal support thresholds again the timing for the loose semantics increases drastically and in this case is off the chart as the experiment was terminated. In general, we can conclude a similar behaviour as for the Student database, but the pattern explosion and timing is worse due to a larger number of attributes and higher connectivity.

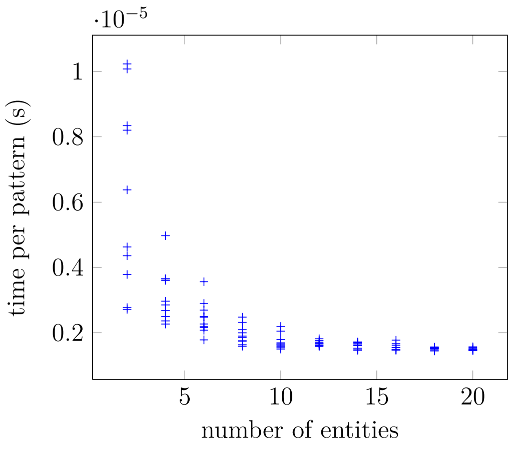
4.5.5 Scalability

To test the scalability of our algorithm we ran SMuRFIG on a collection of synthetically generated databases, with a varying number of entities, tuples, and attributes per entity. These databases were generated as follows[3]. For a given number of entities, a schema is created such that it has a tree shape. Each entity table has a number of attributes that is randomly chosen from a given interval, and each of these attributes has a random number of possible values, drawn from a given interval. The number of tuples per entity is also uniformly picked from a given interval. The entity tables are generated using a Markov chain, i.e. each attribute is a copy of the previous attribute with some given probability, otherwise

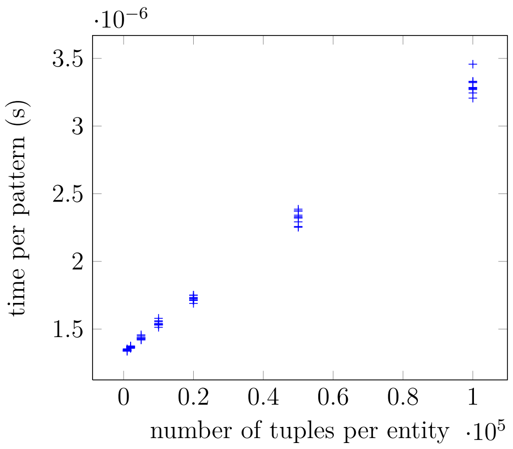
[3] The Python code used to generate the synthetic relational databases can be downloaded at `http://www.adrem.ua.ac.be`.



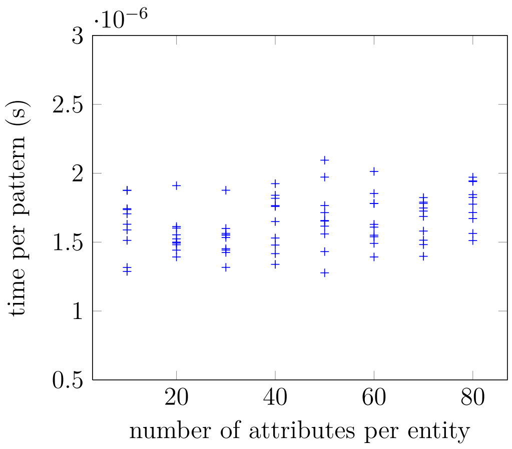
(a) Entity Scalability



(b) Entity Scalability



(c) Tuple Scalability



(d) Attribute Scalability

Figure 4.8: Scalability experiments using synthetically generated relational databases

its value is chosen uniformly at random. The binary relation tables which connect the entities can be seen as bipartite graphs, which are instantiated with a random density drawn from a given interval. These graphs are generated such that the degree distributions of the columns obey a power law[4]. The exponents of these power laws can be computed directly from the density.

For our experiments we chose the number of attributes per table to lie between 5 and 10, the number of values per attribute between 10 and 20, the number of tuples per entity between 10^3 and 10^4, and the relation density between 0.01% and 0.02%. The minimum support threshold for SMuRFIG was set to 1%.

In Figure 4.8a we see that the time spent per frequent itemset increases linearly with the number of entities in the database. However, in Figure 4.8b we see that the time per pattern (i.e. the average time spent for all candidate patterns) *decreases* as the number of entities grows, while the complexity analysis in Section 4.2 says that it should increase, since more intersections and propagations are performed. This apparent contradiction can be explained by the fact that in our analysis we consider the largest size of a KeyID list for some entity E to be $|E|$, while in reality they are often much smaller. In our experiments the average size of the KeyID lists decreases so fast, that intersecting or propagating them actually takes less time. If we were to implement SMuRFIG with, say, bitvectors of constant size, then we would see an increase in the time per pattern. Figure 4.8c shows that as we increase the number of tuples per entity, the time required by SMuRFIG increases linearly, which was predicted. Finally, in Figure 4.8d we see that the number of attributes has no effect on the time per pattern. The number of attributes only increases the number patterns, not the time it takes to compute the support of a pattern.

4.5.6 Conclusion

Overall we can conclude for both datasets that for high support thresholds the loose semantics has its merits in execution time without a large overhead in additional patterns and that the strict semantics clearly is the best option overall. Furthermore, our redundancy measures effectively reduce the number of patterns. We find that minimal divergence pruning is a sensible option to reduce the output if complete results are not desired. Moreover, using deviation as an additional interestingness measure we successfully eliminate uninteresting patterns, creating a concise set of interesting patterns.

[4] We conjecture that many realistic databases obey a power law to some extent, i.e. many tuples have a few connections and a few tuples have many connections.

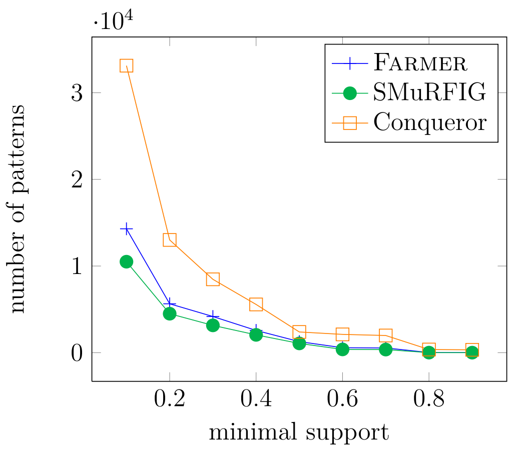
4.6 Related Work

Our work is related to more general frequent query mining approaches such as Warmr [Dehaspe & Toivonen, 1999] and Conqueror (see Chapter 3). The pattern types considered there are much more general and complex. Furthermore, different support measures are used. As such a direct and fair comparison cannot easily be made. We will, however, try to discuss the relationships between these approaches.

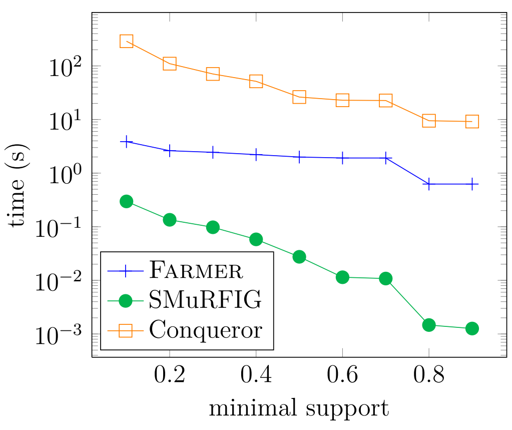
We compare our algorithm to Farmer [Nijssen & Kok, 2003a], which is a variant of the Warmr algorithm that can take into account primary key constraints. For a more detailed discussion of Warmr and Farmer, we refer the reader to Section 3.8 of Chapter 3. In order to be able to compare, we need to restrict the Farmer algorithm to our more specific pattern class, and then run the algorithm several times, each time using a different key, to obtain the same results as SMuRFIG. Doing this adds quite some overhead. On the one hand for the algorithm as it needs to read the data multiple times, on the other hand for a user in preprocessing and postprocessing. To make Farmer generate roughly the same kinds of patterns, we first need to specify how queries need to be constructed for every possible key we want to consider. In the case of our student database, this results in three different input files, one for Student, one for Course and one for Professor. Additionally, for each of these cases different absolute support values needed to be specified, since SMuRFIG uses support relative to each key.

Farmer makes use of object identity, which allows to consider key-constraints. However, it also results in an inequality constraint on all variables. This normally entails that one cannot consider properties of objects, since for such variables an inequality constraint is undesirable. However, Farmer also supports weak object identity [Nijssen & Kok, 2003b], allowing to explicitly specify which attributes should satisfy the inequality constraint. This allows us to consider our desired attribute *sets*, where attributes not present will be 'wildcards' attributes. Unfortunately, this approach in combination with the primary key constraints does not allow us to consider values for keys of an entity. As a workaround we duplicate the key attribute in the dataset presented to Farmer. It is clear that it is not a trivial task to let Farmer generate our pattern class. Although the complex specifications of Farmer allow for great flexibility and the addition of constraints, the amount of input for our algorithm is relatively small while still resulting in patterns that are easy to interpret and expressive. Furthermore, Farmer will also generate more patterns than SMuRFIG, essentially patterns containing only wildcards, e.g. the amount of professors connected to **a** course. There is no way to disable these patterns, since they are required in the generation process.

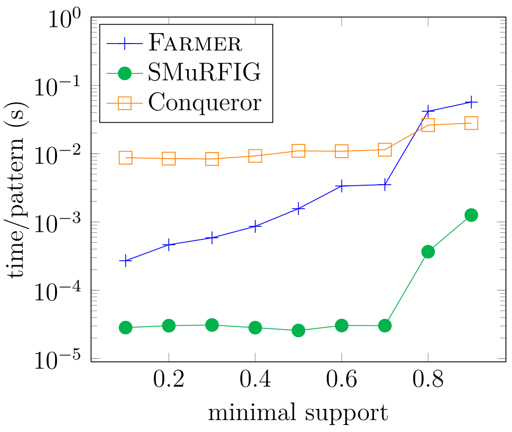
In Figure 4.9a we can see that Farmer and SMuRFIG roughly generate the same number of patterns, although the difference becomes larger for smaller support values. In Figure 4.9b, we can see that SMuRFIG performs better than



(a) Number of patterns



(b) Runtime



(c) Time per pattern

Figure 4.9: Comparison of FARMER, SMuRFIG and Conqueror on Student DB

FARMER. It is, however, not a big difference for this small dataset. We could argue that FARMER generates more patterns, but if we look at the time spent per pattern in Figure 4.9c we can still see that SMuRFIG is better, although it must be noted that the difference becomes smaller for smaller minimal support values. We must additionally mention that we did not include input processing in the timing figure, since FARMER performs this task three times and on flat files (in general the number of keys times), whereas SMuRFIG performs this task once and directly on a relational database. Including this in the timing would result in even more incomparable timing values.

Another WARMR approach based algorithm is the RADAR algorithm [Clare et al., 2004]. This algorithm focusses on reducing the memory requirement compared to the existing approaches. It makes use of an inverted index structure on a flattened version of the database. While it achieves its goal and therefore requires significantly less memory than Warmr and Farmer, it performs poorly on timing for smaller datasets. The SMuRFIG implementation stores the complete database in main memory similar to Farmer. This is mainly for efficiency reasons. However, it is no requirement, and actually only the relations are needed in memory to perform the KeyID transformations. Unlike Farmer we do not keep the found patterns in main memory. Farmer (like Warmr) even keeps some infrequent patterns in order to eliminate equivalent queries. In Section 4.2.2 we already showed that the memory complexity of SMuRFIG is $\mathcal{O}((\frac{n^2}{2}+\frac{n}{2})\cdot(\sum_{E\in\mathcal{E}}|E|)+\sum_{R\in\mathcal{R}}|R|))$. This ensures that SMuRFIG, like Radar, also does not have the same main-memory scalability constraints of Farmer and Warmr.

The frequent query mining algorithms that consider multiple keys [Goethals & Van den Bussche, 2002, Goethals et al., 2008] typically consider every possible set of attributes as a key. We compare SMuRFIG to Conqueror [Goethals et al., 2008] the algoritm presented in Chapter 3. As input to the Conqueror algorithm we need to specify a certain set of attributes of which the subsets can be used for support counting. This has as a result that the algorithm, when provided with the set of keys, any of the subsets of this keyset is considered. For instance in the case of the student database, we can provide {S.SID,C.CID,P.PID}. Conqueror will then also consider for instance {S.SID,C.CID} and {S.SID,P.PID}, while this is not the case in SMuRFIG, where only {S.SID},{C.CID} and {P.PID} are considered separately. This adds a lot of overhead compared to SMuRFIG, certainly when the set of keys is large. This behaviour of Conqueror cannot be disabled, and therefore the comparison is again not completely fair. In Figure 4.9a we can see that for higher minimal support values Conqueror and SMuRFIG roughly generate the same amount of patterns. When looking at the timing in Figure 4.9b, we see that SMuRFIG clearly outperforms Conqueror. However, it must be noted, that Conqueror performs a query in the database for each considered pattern (with some optimisations), a large percentage of time is thus spent in input/output processing, which cannot be trivially excluded as with SMuRFIG or Farmer. This is another reason why we cannot really consider this a fair comparison. This is clear in Figure 4.9c where we can see the time spent per pattern is considerably higher than the other algorithms.

Other more general approaches include tree mining and graph mining [Hoekx & Van den Bussche, 2006, De Knijf, 2007], which can be considered as data-structure-specific special cases of the general query mining approach. Frequently, adapted query mining techniques are therefore used for this purpose. Another specific case of the general query mining approach, is that of multidimensional association rules [Kamber et al., 1997]. In essence multidimensional association rules consist of queries mined only in the key of the central table, the fact table, of a star-scheme (the typical scheme used for data warehouses). If one only considers the so called interdimensional association rules, that do not allow repetition of the predicates (*i.e.* dimension tables), this case of multidimensional association rules roughly corresponds to our definition of rules over relational itemsets, if we would only consider the key of the fact table, and limit ourselves to star-schemes.

As stated, these query mining approaches are more general, and we are not aware of many approaches considering the more specific case of (categorical) itemsets in a multi-relational setting. There is the compression based R-KRIMP [Koopman & Siebes, 2008] algorithm and the algorithm of [Crestana-Jensen & Soparkar, 2000]. Both algorithms use the number of occurrences of the itemset in the join of all tables as the frequency measure. This is done without fully computing, or at least without storing, the join of all tables. Itemsets are basically computed using standard techniques; [Crestana-Jensen & Soparkar, 2000] is based on Apriori [Agrawal et al., 1993] and R-KRIMP is based on KRIMP [Siebes et al., 2006], both with the addition of some case specific optimisations. These algorithms return the same results as Apriori (∼KRIMP) run on the join of all tables. This is not the case for our approach since multiple keys are used as frequency counting measure. Furthermore, compared to the two-phased approach of Crestana-Jensen et al. the computation of intra-entity and extra-entity itemsets is merged in our algorithm, and performed in a depth-first manner. [Ng et al., 2002] focus on star schemes, in which case the join table is in essence already materialised as the fact table. The approach closely mirrors the two-phased approach of Crestana-Jensen et al. but adds some additional optimisations specific for star schemes, and obviously relates to the mentioned multidimensional association rule mining.

We repeat that none of these relational itemset approaches take into consideration the semantic consequences of the blow-up that occurs in the joining of all tables. If for example we have a professor with name `Susan` that teaches a course that almost all students take, in the join of all tables a lot of tuples include the attribute (P.Name,`Susan`). Thus $\{(\mathsf{P.Name},\mathtt{Susan})\}$ represents a frequent relational itemset. Does this mean that a large percentage of professors is named `Susan`? On the contrary, only one is. All of the mentioned relational itemset mining approaches return the itemset as frequent with respect to the join of all tables, which is not that useful. In our approach only $\{(\mathsf{P.Name}, \mathtt{Susan})\}_{\mathsf{S.SID}}$ is a frequent itemset, immediately giving the information of what aspect of this itemset determines its frequency.

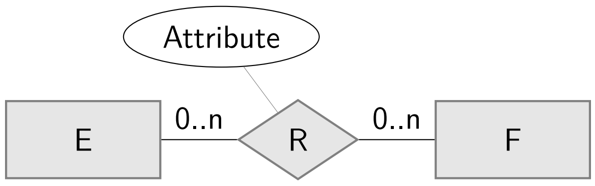
Cristofor and Simovici [Cristofor & Simovici, 2001] do define a support measure similar to ours called *entity support*. For those itemsets consisting of attributes from one entity, next to the join support, they also consider the support in the number of unique tuples of that entity. In other words, they do consider KeyID based support, but only for inter-entity itemsets and only in the key of that entity. They are therefore unable to express a pattern like $\{(\mathsf{P.Name} = \mathtt{Susan})\}_{\mathsf{S.SID}}$. Furthermore, their proposed Apriori based algorithm, which is defined only for star-schemes, explicitly computes joins, which as we already stated does not scale well for larger databases.

The importance of considering multiple keys is also reported by [Koopman & Siebes, 2009]. They developed the RDB-Krimp algorithm that selects a subset of possible patterns that best describes (compresses) the database. They show that using patterns over multiple keys allows for a better description of the database, even in the case of a database with a clear centralised (one-target) structure. Algorithms, like ours, which efficiently generate patterns in multiple keys are therefore complementary to this approach.

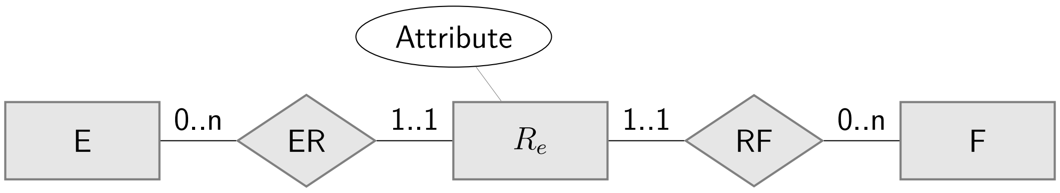
4.7 Conclusion

In this chapter we generalised itemset mining to a (multi-)relational setting by defining frequent relational itemsets using a novel support measure based on the keys of the relational scheme. We implemented an efficient propagation-based depth-first algorithm for mining these frequent relational itemsets and confident relational association rules. Our experiments showed that the SMuRFIG algorithm performs well on real datasets and is capable of finding interesting patterns. Although the naive approach we also proposed is feasible for small datasets, it has no real advantages. We considered two different support semantics, the loose and the strict semantics. The experiments showed that for high support thresholds SMuRFIG with loose semantics has its merits in execution time without a large overhead in additional patterns, but that SMuRFIG using strict semantics is the best option overall. In addition, we defined the deviation measure to address the statistical pattern blow-up that is specific to the relational context, and we experimentally showed that it works as promised. Furthermore, we generalised some popular redundancy measures - closure-based redundancy and minimal improvement - to our multi-relational setting, and confirmed that they can reduce the output if complete results are not desired.

Our initial pattern and scheme definition already allows us to mine a large collection of databases (or parts of databases) and results in interesting patterns found, as can be seen in the Section 4.5. Preliminary experiments already show that some small extensions to our definitions, allow us to find complex patterns while only suffering a relatively small loss of efficiency. These extensions are subject of Section 4.8.



(a) Relation with attributes



(b) Simulated Relation Attributes

Figure 4.10: Relation Attribute Simulation

4.8 Further Research

In this chapter, we restricted ourselves to simple schemes for clarity and efficiency. The simple scheme setting already allows us to mine a large collection of databases (or parts of databases) and results in interesting patterns found, as can be seen in previous sections. Some small extensions to our definitions, however, allow us to find complex patterns while only suffering a relatively small loss of efficiency. In this section we show how some of these extension can be simulated in SMuRFIG or which changes to the algorithm would be required.

4.8.1 Relation Attributes

SMuRFIG only considers attributes as parts of entities. But in general it is also possible that relations have attributes. As an example we can consider a typical labeled tree structure. This can be modeled in a relational database by storing nodes as entities and the edges as relations. The labels would then be stored as relation attributes.

Relation attributes can be simulated in our current technique by changing the database scheme by introducing entities for each relation. Consider for example an entity E connected to an entity F using relation R which has an attribute, shown in Figure 4.10a. This scheme could be transformed in entities E, R_e, F and relations ER and RF respectively connecting entity E and R_e, and R_e and F shown in Figure 4.10b. Every connection in R is now split up into a connection of E to a unique tuple in R_e contained in ER and a connection of the unique tuple in R_e to F in RF. The attributes of the tuples in relation R are now attributes of unique tuples in the entity R_e and can therefore be mined using our technique. This also gives an added type of pattern, namely patterns counted in the key of R_e, which would represent frequency in the number of connections between E and R. Support for relation attributes is therefore possible in SMuRFIG, without any additional changes.

Of course this simulation entails some overhead. It is, however, possible to adapt the SMuRFIG algorithm to be able to mine relation attributes directly. First of all, next to the entities, we need to mine singleton itemsets in the relations. As key for these itemsets we take the pair of keys of the entities the relation connects. These singleton items row-id frequencies also need to be translated to every possible key of other entities and relations. So in essence the loop in SMuRFIG should just consider all R next to all E. Similarly in Keyclat, relations should be treated in the same way as entities with a defined key, and propagation from these new keys should be added in Propagate. This way relation attributes would be able to be found efficiently. Note, that we do not consider including itemsets frequent in relation-keys (essentially frequent in a set of two entity keys) in our output, since we only want to consider entitiy keys for our final support measure.

4.8.2 Wildcard Patterns

The current pattern definition for SMuRFIG is limited to connected (attribute, value) pairs. In this setting, for example, we cannot express the pattern "all professors named Jan that teach **a** course". This obstacle can be resolved without extensively expanding the pattern syntax, namely by introducing *virtual attributes*. We could assume that every entity has a virtual attribute with the same value for each tuple (e.g '`true`').

Example 4.14. *Given the scheme of Figure 4.2, next to the itemset* $\{(\mathsf{P.Name} = \mathsf{Jan})\}_{\mathsf{P.PID}}$ *we can now also consider the itemset* $\{(\mathsf{P.Name} = \mathsf{Jan}), (\mathsf{C.Course} = \mathtt{true})\}_{\mathsf{P.PID}}$ *which expresses "all professors named* **Jan** *that teach* **a** *course". Here,* $(\mathsf{C.Course} = \mathtt{true})$ *is the virtual attribute, representing any course. More than that, association rules like* $\{(\mathsf{C.Course} = \mathtt{true})\} \Rightarrow_{\mathsf{P.PID}} \{(\mathsf{S.Study} = \mathtt{1MINF\text{-}CN})\}$ *are possible. This rule, which we found in preliminary experiments, had a confidence of 63%, meaning that this percentage of the professors teaching a course, teach a course followed by students of the study 1MINF-CN. Unfortunately next to the itemsets like* $\{(\mathsf{C.Credits} = 10)\}_{\mathsf{P.PID}}$*, we now also find* $\{(\mathsf{C.Credits} = 10), (\mathsf{C.Course} = \mathtt{true})\}_{\mathsf{P.PID}}$*. In fact* $(\mathsf{C.Course} = \mathtt{true})$ *is in the closure of any itemset containing a Course attribute.*

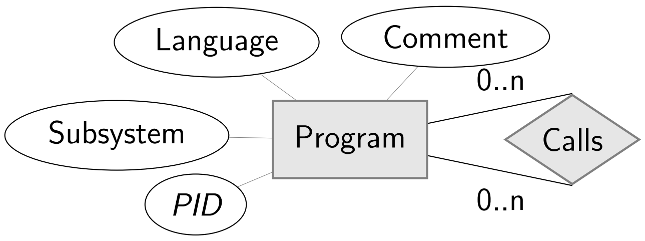


Figure 4.11: Example Inherently Cyclic Scheme

In order to generate these kinds of patterns we can really add an attribute to every entity containing the same value for every tuple. For example we can add a 'Course' attribute to the Course entity with value '`true`' for every tuple in the relation. This allows the current SMuRFIG algorithm to generate these kinds of patterns. However, since we know these are no ordinary attributes it would be more efficient to include the generation inside the algorithm. The singleton virtual attributes can easily be added in the *singletons* procedure since we know their support is $|E|$. As for the combinations in Keyclat, we can disallow any combination of two itemsets where one contains a virtual attribute and the other contains an attribute of the same relation the virtual attribute belongs to. These itemsets would be redundant with respect to the non-closed variant. Note, that when combining these virtual attributes with the previously introduced relation attributes, we can also consider virtual relation attributes.

4.8.3 Simple Cycles

Some simple relational data is already inherently cyclic. Consider for instance a Software Engineering dataset of which the scheme is given in Figure 4.11. We have one entity, Program, which stores information about programs like their subsystem, language, comments etc. Next to this entity we have one relation, Calls, which directionally connect programs that call each-other. Our current approach is not able to handle this setting for the simple reason that we only consider every entity (and relation) once. To express a pattern about one program calling another one would at least require two instances of the Program entity. A simple way to simulate this behaviour in our current approach is to create a copy of the program entity, which we could call ProgramCalled for instance. This way we can express patterns about one Program calling another, counted in the *calling* program as well as the *called* program. An unfortunate side effect is that patterns only considering attributes of the Program entity counted in its key will be found twice; once for Program and once for ProgramCalled. To mitigate this problem, we could allow to specify *virtual copies* of entities. For virtual copies we would not generate intra-entity itemsets, ensuring these patterns are only be generated once. Furthermore, we do need to consider the virtual copy's key and attributes in combination with other entities. We must note, that if one wants to consider longer cycles, for example, programs called by programs called by programs, one needs to create multiple virtual copies, and when the entity is connected to other entities, these too must be copied. It is clear that this solution is far from optimal, and further study is required. It can, however, provide a workable approach in simple cases.

Furthermore, we must note that these simple cycles differ from general cyclic patterns. We consider simple cycles as being induced by an inherently cyclic scheme. One could also consider cycles in non-cyclic schemes. For example we could consider patterns concerning students and other students in the same study. This kind of pattern, if expressed as a query, would require two occurrences of the Student entity and two occurrences of the Studies relation.

Example 4.15. *For example, the pattern* $\{(\mathsf{S1.Surname} = \mathsf{A}), (\mathsf{S2.Surname} = \mathsf{LP})\}_{\mathsf{Y.YID}}$ *counts the amount of studies in which there is a student with surname* A *as well as one with surname* LP*. In order to evaluate frequency one would need to write a query like*

$$\pi_{Y.YID}\sigma_{S_1.SID=St_1.SID, S_2.SID=St_2.SID, Y.YID=St_1.YID=Sty_2.YID} S_1 \times S_2 \times St_1 \times St_2 \times Y$$

where it is clear that two copies of the Student *entity (S) and* Studies *relation (St) are required.*

Although a technique similar to the virtual copies could be used to accomplish this, it is clear that the limits here are not clearly defined, and further investigation of this issue is needed.

4.8.4 Graph Schemes

The current SMuRFIG approach is based on simple relational schemes. When we drop point 4.1 of the definition of a simple relational scheme (Definition 4.1), then two different entities are allowed to be connected in multiple ways, *i.e.*, we extend tree-schemes to more general graph-schemes. One could for instance imagine that we expand the scheme of Figure 4.2 with an additional relation Thesis which connects Professors and Students, as shown in Figure 4.12. In this way, students can either be connected with a professor via Thesis or via a Course (actually Takes and Teaches). Under this generalisation, we would need to reconsider our support definition as it is based on the existence of a unique path. We can consider two options:

1. All attributes are connected in all possible ways

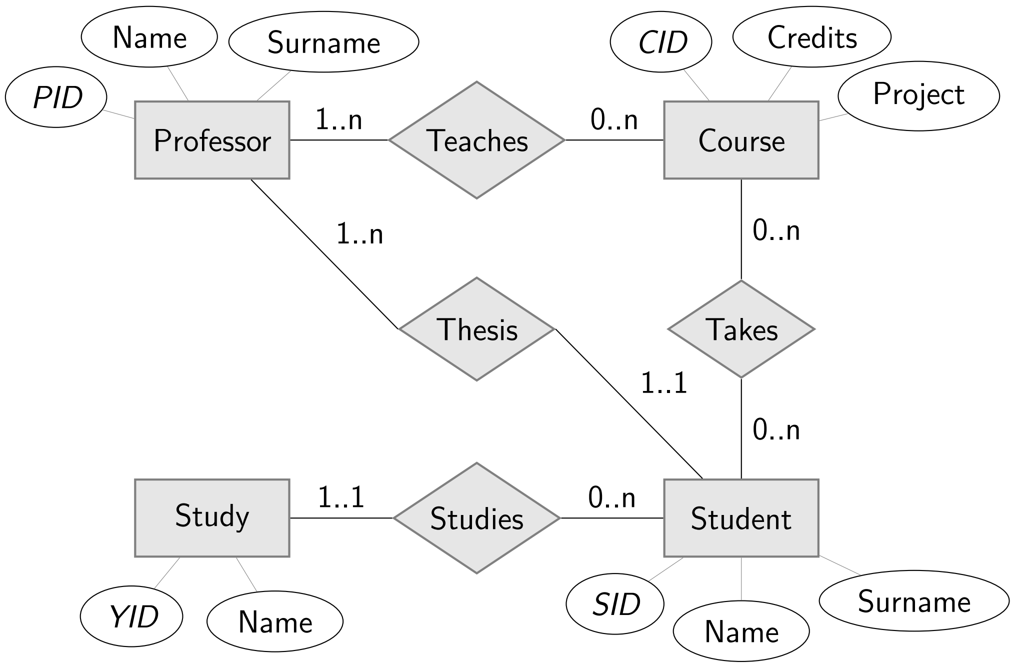


Figure 4.12: Example Graph Relational Scheme

2. All attributes are connected in any of the possible ways

Depending on the choice, we have a different semantics for our patterns. Option 1 is very strict. For instance, in the example of the added Thesis relation this would imply we could only express patterns about students and their thesis supervising professor, since a pattern with another professor is not valid. Option 2 is very loose, if we have a pattern about students and professors we know that they are connected in **a** way, which might not be very useful for someone trying to interpret the patterns. There is, however, a third option:

3. All attributes are connected in the specified way

This specified way, could then be the way specified in the itemset itself. Similar to the virtual attributes introduced to support wildcard patterns, we can consider virtual relation attributes to consider different paths. The semantics of support for such itemsets can be clearly defined as the answer of the query using the path of the specified relations.

Example 4.16. *The support of the pattern*

$$\{(\mathit{S.Surnname} = \mathit{VG}), (\mathit{Takes} = \mathit{Y}), (\mathit{Teaches} = \mathit{Y})\}_{\mathit{P.PID}}$$

would be computed using the query

$$\pi_{P.PID}\sigma_{S.Surname=\text{‘}VG\text{'}} S \bowtie_{SID} Takes \bowtie_{CID} Teaches \bowtie_{PID} P$$

while that of pattern $\{(S.Surname = VG), (Thesis = Y)\}_{P.PID}$ *would be computed using*

$$\pi_{P.PID}\sigma_{S.Surname=\text{‘}VG\text{'}} S \bowtie_{SID} Thesis \bowtie_{PID} P$$

If one also allows relational itemsets without any virtual relation attributes occurring, one has the full range of patterns: those that are connected in any way (no virtual relation attributes), those connected in a specified way (some virtual relation attributes present) and those connected in all possible ways (all possible virtual relation attributes present). Only the first one needs an additional definition of support, but it is clear that this support can be computed by using the union of all the queries of all possible paths for the relational itemset. For instance, the support of the itemset $\{(S.Surname = VG)\}_{P.PID}$ would be computed using the union of the two queries given in Example 4.16. However, since these patterns with no relation-attributes are essentially an aggregation of more specific patterns described using virtual relation-attributes, these patterns can always be considered using postprocessing.

In order to generate relational itemsets with specified virtual relation attributes, the *translate* subroutine needs to be modified, similar to the modification for the addition of virtual attributes, to add the different virtual relation attributes. In the graph scheme case, however, a singleton itemset can be translated to one key in more than one way, thus resulting in multiple itemsets.

Example 4.17. *The singleton relational itemset* $\{(S.Surname = VG)\}$ *can be translated to* $P.PID$ *in three ways:*

- $\{(S.Surname = VG), (Takes = Y)(Teaches = Y)\}_{P.PID}$
- $\{(S.Surname = VG), (Thesis = Y)\}_{P.PID}$
- $\{(S.Surname = VG), (Takes = Y)(Teaches = Y)(Thesis = Y)\}_{P.PID}$

The KeyID list of the last one can be computed by intersecting the KeyID lists first two.

Unfortunately, the computation of other inter-entity itemsets now becomes more difficult because one needs to take into account all the different paths. The changes necessary in the Keyclat algorithm when the scheme represents a graph, are located in the propagate function. Since now there may be several possible paths between any two entities, propagation from one entity to another can be done in several ways. It is clear that this makes computations more complex, but specific knowledge of the topology of the scheme could be used to optimise the algorithm. Since an optimal solution is not straightforward, this generalisation remains further work.

Chapter 5

Conclusions and Further Research

This dissertation introduced new algorithms for mining interesting patterns in relational databases. We investigated the theoretical as well as the practical aspects of these algorithms. In this chapter we summarise our main contributions, and provide some directions for further research.

5.1 Summary

The Information Age has provided us with huge data repositories which cannot longer be analysed manually. The potential high business value of the knowledge that can be gained, however, drives the research for automated analysis methods that can handle large amounts of data. Most of the data of industry, education and government is stored in relational database management systems (RDBMSs). This motivates the need for data mining algorithms that can work with arbitrary relational databases, without the need for manual transformation and preprocessing of the data. To provide in this need, this dissertation introduced efficient data mining algorithms that can find *interesting patterns* in *relational databases.*

We first explained the general problem of frequent pattern mining. Frequency is a basic constraint in pattern mining. Since data mining typically deals with huge volumes of data, in order for a pattern to be interesting it must hold for a large portion of this data. Hence it must occur frequently. We first looked at frequent itemset mining, the simplest variant of frequent pattern mining. It provided an introduction to the basic properties and techniques that are also needed when considering mining more complex pattern kinds. We looked at the most important algorithms, Apriori and Eclat, as well as how to avoid redundancy using closure. Furthermore, we considered how techniques of frequent itemset mining can be generalised to frequent pattern mining. We provided a short overview of the pattern kinds that have been studied in the field of relational database mining, and introduced the two which this dissertation examines more thoroughly: queries and relational itemsets.

We introduced *queries* as a pattern kind to be mined in relational databases. We proposed a new and appealing type of association rules, by pairing *simple conjunctive queries*. We presented our novel algorithm *Conqueror*, that is capable of efficiently generating and pruning the search space of all simple conjunctive queries, and we presented promising experiments, showing the feasibility of our approach, but also its usefulness towards the ultimate goal of discovering patterns in arbitrary relational databases. Next to many different kinds of interesting patterns, we have shown that these rules allow us to discover *functional dependencies*, *inclusion dependencies*, but also variants thereof, such as *conditional* and *approximate* dependencies. We related confidence as a measure to the existing measures developed for approximate functional and inclusion dependencies and showed that Conqueror mines both pattern types at the same time.

After this, we extended our basic approach in order to efficiently mine relational databases over which *functional dependencies* are assumed. By using these provided functional dependencies, we are able to prune the search space by removing the redundancies they cause. Since we already showed that our algorithm was capable of detecting functional dependencies that were not given initially, we then extended our algorithm even further, such that it also uses newly discovered, previously unknown functional dependencies to prune even more redundant patterns. Moreover, since not only the functional dependencies that hold on the database relations are discovered, but also functional dependencies that hold on joins of relations, these too are subsequently used to prune yet more queries. Besides functional dependencies, we also adapted our algorithm to detect and use foreign keys, since they give rise to redundant evaluations. We implemented and tested our updated algorithm, *Conqueror*$^+$, and showed that it efficiently reduces the number of queries that are generated and evaluated, by detecting and using functional dependencies and foreign keys even when none are provided, and that it potentially greatly outperforms the previous algorithm in time. As such, the Conqueror$^+$ algorithm provides us with a method to efficiently discover a concise set of interesting patterns over arbitrary relational databases.

Next to queries, we also investigated another simple, but powerful class of patterns: *relational itemsets*. Relational itemset mining is a generalisation of itemset mining to the (multi-)relational setting. We defined frequent relational itemsets using a *novel support measure* based on the keys of the relational scheme. We implemented a novel propagation-based efficient depth-first algorithm, called *SMuRFIG* (Simple Multi-Relational Frequent Itemset Generator), to mine these frequent relational itemsets and confident relational association rules. In addition we defined the *deviation* measure to address the statistical pattern blow-up intrinsic to the relational case, and showed that it works as promised. Furthermore, we concluded that our generalisation of closure to the relational itemset case works well, as well as the novel notion of minimal divergence pruning, which is shown to be a sensible option to reduce the output if complete results are not required.

5.2 Further Research

The field of mining arbitrary relational databases still provides many challenges. Although our approaches provide algorithms that are capable of efficiently discovering interesting patterns, there are several ways in which they could be improved.

The greatest bottleneck for the Conqueror and Conqueror$^+$ algorithms is the database communication. Although using SQL provides an algorithm that is applicable to any available RDBMSs, for efficiency reasons it is clear that integrating the algorithm into a specific RDBMS would be a path to consider. Such integration would allow Conqueror to access the data structures directly and have its 'queries' performed as optimally as possible.

Next to these efficiency improvements, there are also many interesting research opportunities in extending the pattern language. Although we showed that simple conjunctive queries are capable of expressing many types of interesting patterns, especially when considering association rules, there is room for improvement. The restriction that every relation is only allowed once in the join prohibits the expression of patterns of the type "70% of the students that follow class A also follow class B". Dropping this restriction, however, opens up the floor to new problems regarding cyclicality of graphs and query equality checking. As we already mentioned, the problem of checking equality for general conjunctive queries and the problem of checking graph isomorphism are both hard. Further study is therefore vital in order to find interesting subclasses for which scalable and efficient algorithms can be developed.

The nature of relational data, *i.e.* many entities connected to many other entities, inherently result in even more problems in generating a *concise* set of patterns than in the classical case, where only one entity is considered. In the Conqueror$^+$ algorithm we detected and applied functional dependencies to be able to present the pattern set more concisely. We also touched on the fact that approximate functional dependencies result in many ‘redundancies’ as well. However, we concluded that these could not be eliminated if complete results are required. The investigation of *approximate results* and the potential application of approximate functional dependencies in doing so, is an interesting topic that merits further research. Furthermore, we generalised our query comparison to only take functional dependencies into account. As stated, combining this with inclusion dependencies is not feasible in the general case, and we therefore only used foreign-keys knowledge to avoid query evaluation. Further research is required to identify specific restricted cases where it is possible to define query equivalence in such a way that it exploits both, as done in the case of key-foreign-key joins in star-schemes [Jen et al., 2009].

In the setting of relational itemset mining we already covered the potential for further research in Section 4.8. We showed that extending the pattern definition to allow wildcards and for relations to have attributes, can easily be accomplished. Dropping restrictions on the scheme to allow cycles and in general graphs shaped schemes is, however, non trivial and requires a radically different mining strategy. Such an elimination of restrictions corresponds to the extension of the pattern class in the setting of conjunctive queries mentioned above. In order to achieve such goals, techniques used in graph mining should be investigated, although most focus in that domain has been directed towards mining structural properties. One approach one could consider, is to first apply graph mining methods to extract interesting structures and then perform (relational) itemset mining techniques on the results in order to find attribute-value related patterns.

It is clear that the field offers many possibilities for further research and that many more insights can be gained by applying data mining to all kinds of relational databases used in the world today.

Nederlandse Samenvatting

DATAGENERATIE- en opslagtechnologieën hebben de afgelopen decennia een sterke groei gekend. Deze explosieve groei van beschikbare informatie maakt dat we dringend nood hebben aan technieken om deze grote hoeveelheden data te analyseren. Het grote volume maakt manuele analyse onhaalbaar, waardoor analyse met behulp van intelligente algoritmes een onmisbaar instrument is in moderne data analyse. *Data mining*, of ook wel *knowledge discovery from data (KDD)*, is 'de automatische extractie van patronen die interessante kennis representeren impliciet opgeslagen in grote databases, het web of andere grote databronnen'. Het relationele data model is de laatste decennia het dominante paradigma voor industriële database applicaties. Het is de basis van de grootste commerciële database systemen, waardoor *Relational Database Management Systemen (RDBMSen)*, de meest voorkomende soort data repository zijn. De populairste taal om RDBMSen te ondervragen is de Structured Query Language (SQL). *Het doel van deze thesis is het ontwerp van efficiënte data mining algoritmes die interessante patronen kunnen ontdekken in relationele databases.*

In grote hoeveelheden data is een patroon meestal pas interessant als het geldt voor een voldoende groot deel van de data. Dit is de basis van *frequent pattern mining*. De meest eenvoudige en historisch eerst beschouwde vorm van frequent pattern mining is *frequent itemset mining*. De eigenschappen en technieken die werden geintroduceerd in frequent itemset mining vormen ook de basis voor gelijkaardige eigenschappen en technieken die nodig zijn voor het beschouwen van complexere patroontypes. We bekeken dan ook de belangrijkste frequent pattern mining algoritmes, *Apriori* en *Eclat* en hoe overbodige patronen kunnen vermeden worden door het toepassen van *closure*. Vervolgens beschouwden we hoe we deze technieken uit frequent itemset mining kunnen veralgemenen naar het minen van frequente patronen in het algemeen. We besloten met een kort overzicht van de verschillende soorten patroontypes die in het onderzoeksgebied van het minen van relationele databases aan bod komen. Hierbij introduceerden we ook de twee types die we in deze thesis grondiger bestudeerden: *queries en relationele itemsets*.

We introduceerden *queries* als een patroontype om te minen in relationele databases. We stelden een nieuw en aantrekkelijk type *association rules* voor door koppels van twee *simpele conjunctieve queries* als regels te beschouwen. We presenteerden ons nieuwe algoritme *Conqueror* (Conjunctive Query Generator) dat in staat is om efficiënt en zonder duplicaten de zoekruimte van alle simpele conjunctieve queries te genereren. Verder presenteerden we experimentele resultaten die de haalbaarheid van onze aanpak aantoonden maar ook de bruikbaarheid van de gegenereerde patronen. Dit toont aan dat het minen van simpele conjunctieve queries een nuttige aanpak is om interessante patronen te kunnen ontdekken in arbitraire relationele databases.

Naast verschillende andere soorten interesante patronen, toonden we aan dat koppels simpele conjunctieve queries ons toelaten om *functionele afhankelijkheden* en *inclusie afhankelijkheden* te vinden, maar ook varianten zoals *conditionele* en *benaderende* afhankelijkheden. We beschreven hoe *confidence* als maat relateert met de bestaande maten die ontwikkeld werden voor benaderende functionele en inclusie afhankelijkheden en toonden aan dat Conqueror deze beide patroontypes simultaan kan minen.

Hierna breidden we onze basisaanpak uit om efficiënt relationele databases te minen waarin we veronderstellen dat functionele afhankelijkheden aanwezig zijn. Door gebruik te maken van deze functionele afhankelijkheden zijn we in staat om de zoekruimte te verkleinen door overbodigheden te verwijderen die door deze afhankelijkheden worden veroorzaakt. Omdat we reeds aantoonden dat het Conqueror algoritme in staat is om voorheen onbekende functionele afhankelijkheden te ontdekken, breidden we ons algoritme nog verder uit zodanig dat het ook nieuw ontdekte functionele afhankelijkheden gebruikt om overbodige queries niet te genereren. Omdat we bovendien ook functionele afhankelijkheden kunnen ontdekken die op *joins* van relaties gelden, pasten we ons algoritme aan om ook deze te gebruiken om nog meer overbodige generaties te vermijden. Hiernaast voegden we ook de mogelijkheid om *foreign keys* te detecteren en te gebruiken aan ons algoritme toe, want ook deze geven aanleiding tot overbodige query evaluaties. Al deze aanpassingen resulteerden in een aangepast algoritme *Conqueror*$^+$, waarvan we aantoonden dat het efficiënt het aantal overbodige queries reduceert door de detectie en het gebruik van functionele afhankelijkheden en foreign keys. Door deze optimalisaties is het Conqueror$^+$ algoritme dan ook beduidend sneller dan het vorige Conqueror algoritme. Conqueror$^+$ is dus een efficiënte methode om een beknopte set van interessante patronen te ontdekken in arbitraire relationele databases.

Naast queries bestudeerden we een tweede simpele maar tegelijk ook krachtige klasse van relationele patronen: de *relationele itemsets*. Relationele itemset mining is een veralgemening van itemset mining naar de relationele setting. We definieerden frequente relationele itemsets gebruik makend van een nieuwe support maat gebaseerd op de keys van het relationele schema. We implemteerden het nieuwe efficiënt propagatie gebaseerd depth-first algoritme *SMuRFIG* (Simple Multi-Relational Frequent Itemset Generator) om deze frequente relationele itemsets en bijhordende confidente relationele associatie regels te minen. Hier bovenop definieerden we de *deviation* maat om de statische *blow-up* van patronen die voorkomt in het relationele geval tegen te gaan. We toonden experimenteel aan dat deze maat zijn werk doet. Verder concludeerden we dat onze veralgemening van closure naar de relationele setting goed werkt, net als dat de nieuwe notie van mininmal divergence – die kan gezien worden als een veralgemening van mininal improvement – een goede optie is om de resultaat set te verkleinen als complete resultaten niet vereist zijn.

Bibliography

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Agrawal et al., 1993] Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In P. Buneman & S. Jajodia (Eds.), *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22:2 of *SIGMOD Record* (pp. 207–216).: ACM Press.

[Agrawal et al., 1996] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., & Verkamo, A. I. (1996). Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining* (pp. 307–328). AAAI/MIT Press.

[Agrawal & Srikant, 1994] Agrawal, R. & Srikant, R. (1994). Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, & C. Zaniolo (Eds.), *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94), September 12-15, 1994, Santiago de Chile, Chile* (pp. 487–499).: Morgan Kaufmann.

[Agrawal & Srikant, 1995] Agrawal, R. & Srikant, R. (1995). Mining sequential patterns. In P. S. Yu & A. L. P. Chen (Eds.), *Proceedings of the 11th International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan* (pp. 3–14).: IEEE Computer Society.

[Baixeries, 2004] Baixeries, J. (2004). A formal concept analysis framework to mine functional dependencies. In *Proceeding of the Workshop on Mathematical Methods for Learning, Villa Geno, Italy.*

[Bastide et al., 2000] Bastide, Y., Pasquier, N., Taouil, R., Stumme, G., & Lakhal, L. (2000). Mining minimal non-redundant association rules using frequent closed itemsets. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, & P. J. Stuckey (Eds.), *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science* (pp. 972–986).: Springer.

[Bayardo Jr et al., 2000] Bayardo Jr, R., Agrawal, R., & Gunopulos, D. (2000). Constraint-based rule mining in large, dense databases. *Data Mining and Knowledge Discovery*, 4(2), 217–240.

[Bell & Brockhausen, 1995] Bell, S. & Brockhausen, P. (1995). Discovery of data dependencies in relational databases. In Y. Kodratoff, G. Nakhaeizadeh, & C. Taylor (Eds.), *Statistics, Machine Learning and Knowledge Discovery in Databases.*

[Bocklandt, 2008] Bocklandt, R. (2008). http://www.persecondewijzer.net.

[Bohannon et al., 2007] Bohannon, P., Fan, W., Geerts, F., Jia, X., & Kementsietsidis, A. (2007). Conditional functional dependencies for data cleaning. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey* (pp. 746–755).

[Boulicaut, 1998] Boulicaut, J.-F. (1998). A KDD framework for database audit. In *Proceedings of the 8th Workshop on Information Technologies and Systems* Helsinki, Finland.

[Brin et al., 2003] Brin, S., Rastogi, R., & Shim, K. (2003). Mining optimized gain rules for numeric attributes. *IEEE Transactions on Knowledge and Data Engineering*, 15(2), 324–338.

[Calders et al., 2007] Calders, T., Goethals, B., & Mampaey, M. (2007). Mining itemsets in the presence of missing values. In Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, & Y. W. Koo (Eds.), *Proceedings of the ACM Symposium on Applied Computing (SAC)* (pp. 404–408).: ACM.

[Casanova et al., 1984] Casanova, M. A., Fagin, R., & Papadimitriou, C. H. (1984). Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1), 29–59.

[Chandra & Merlin, 1977] Chandra, A. K. & Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th annual ACM symposium on Theory of computing (STOC '77)* (pp. 77–90). New York, NY, USA: ACM.

[Chekuri & Rajaraman, 2000] Chekuri, C. & Rajaraman, A. (2000). Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2), 211–229.

[Chiang & Miller, 2008] Chiang, F. & Miller, R. J. (2008). Discovering data quality rules. *Proceedings of the VLDB Endowment*, 1(1), 1166–1177.

[Clare et al., 2004] Clare, A., Williams, H., & Lester, N. (2004). Scalable multi-relational association mining. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004)* (pp. 355–358).

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

[Crestana-Jensen & Soparkar, 2000] Crestana-Jensen, V. & Soparkar, N. (2000). Frequent itemset counting across multiple tables. In *Proceedings of the 4th Pacific-Asia Conference (PAKDD), Kyoto, Japan, April 18-20, 2000* (pp. 49–61).

[Cristofor & Simovici, 2001] Cristofor, L. & Simovici, D. (2001). *Mining Association Rules in Entity-Relationship Modeled Databases*. Technical Report 2001-1, University of Massachusetts Boston.

[Date, 1986] Date, C. J. (1986). *Relational database: selected writings*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[De Knijf, 2007] De Knijf, J. (2007). Fat-miner: mining frequent attribute trees. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007* (pp. 417–422).

[Dehaspe & Raedt, 1997] Dehaspe, L. & Raedt, L. D. (1997). Mining association rules in multiple relations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP '97)* (pp. 125–132). London, UK: Springer-Verlag.

[Dehaspe & Toivonen, 1999] Dehaspe, L. & Toivonen, H. (1999). Discovery of frequent DATALOG patterns. *Data Mining and Knowledge Discovery*, 3(1), 7–36.

[Dehaspe & Toivonen, 2001] Dehaspe, L. & Toivonen, H. (2001). Discovery of relational association rules. In *Relational Data Mining* (pp. 189–208). New York, NY, USA: Springer-Verlag New York, Inc.

[Diop et al., 2002] Diop, C., Giacometti, A., Laurent, D., & Spyratos, N. (2002). Composition of mining contexts for efficient extraction of association rules. In

Proceedings of the 8th International Conference on Extending Database Technology (EDBT'02), Prague, Czech Republic, March 25-27, volume 2287 of *LNCS* (pp. 106–123).: Springer Verlag.

[Džeroski, 2005] Džeroski, S. (2005). Relational data mining. *Data Mining and Knowledge Discovery Handbook*, (pp. 869–898).

[Engle & Robertson, 2008] Engle, J. T. & Robertson, E. L. (2008). HLS: Tunable mining of approximate functional dependencies. In W. A. Gray, K. G. Jeffery, & J. Shao (Eds.), *Proceedings of the 25th British National Conference on Databases (BNCOD 25), Cardiff, UK, July 7-10*, volume 5071 of *Lecture Notes in Computer Science* (pp. 28–39).: Springer.

[Fan et al., 2008] Fan, W., Geerts, F., Jia, X., & Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2).

[Fan et al., 2009] Fan, W., Geerts, F., Lakshmanan, L. V. S., & Xiong, M. (2009). Discovering conditional functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), March 29 2009 - April 2 2009, Shanghai, China* (pp. 1231–1234).: IEEE.

[Flach & Savnik, 1999] Flach, P. A. & Savnik, I. (1999). Database dependency discovery: A machine learning approach. *AI Communications*, 12(3), 139–160.

[Giannella & Robertson, 2004] Giannella, C. & Robertson, E. L. (2004). On approximation measures for functional dependencies. *Information Systems*, 29(6), 483–507.

[Goethals et al., 2005] Goethals, B., Hoekx, E., & Van den Bussche, J. (2005). Mining tree queries in a graph. In *Proceedings of the 11th ACM SIGKDD international conference on Knowledge discovery in data mining (KDD '05)* (pp. 61–69). New York, NY, USA: ACM.

[Goethals et al., 2009] Goethals, B., Le Page, W., & Mampaey, M. (2009). *Mining Interesting Sets and Rules in Relational Databases.* Technical Report 2009-02, University of Antwerp.

[Goethals et al., 2008] Goethals, B., Le Page, W., & Mannila, H. (2008). Mining association rules of simple conjunctive queries. In *Proceedings of the SIAM International Conference on Data Mining (SDM), April 24-26, 2008, Atlanta, Georgia, USA* (pp. 96–107).

[Goethals et al., 2010] Goethals, B., Page, W. L., & Mampaey, M. (2010). Mining interesting sets and rules in relational databases. In *Proceedings of the 25th ACM Symposium on Applied Computing.*

[Goethals & Van den Bussche, 2002] Goethals, B. & Van den Bussche, J. (2002). Relational association rules: Getting WARMeR. In *Proceedings of the Pattern Detection and Discovery: ESF Exploratory Workshop, London, UK, September 16-19, 2002*: Springer.

[Han & Kamber, 2006] Han, J. & Kamber, M. (2006). *Data mining: concepts and techniques.* Morgan Kaufmann.

[Han et al., 2000] Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)* (pp. 1–12). New York, NY, USA: ACM.

[Han et al., 2004] Han, J., Pei, J., Yin, Y., & Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1), 53–87.

[Hoekx & Van den Bussche, 2006] Hoekx, E. & Van den Bussche, J. (2006). Mining for tree-query associations in a graph. In *Proceedings of the 6th International Conference on Data Mining (ICDM)* (pp. 254–264). Washington, DC, USA: IEEE Computer Society.

[Huhtala et al., 1999] Huhtala, Y., Karkkainen, J., Porkka, P., & Toivonen, H. (1999). Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2), 100–111.

[IMDB, 2008] IMDB (2008). http://imdb.com.

[Jen et al., 2008] Jen, T.-Y., Laurent, D., & Spyratos, N. (2008). Mining all frequent projection-selection queries from a relational table. In A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, & I. Manolescu (Eds.), *Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008), Nantes, France, March 25-29*, volume 261 of *ACM International Conference Proceeding Series* (pp. 368–379).: ACM.

[Jen et al., 2009] Jen, T.-Y., Laurent, D., & Spyratos, N. (2009). Mining frequent conjunctive queries in star schemas. In *Proceedings of the 2009 International Database Engineering; Applications Symposium (IDEAS '09)* (pp. 97–108). New York, NY, USA: ACM.

[Jen et al., 2006] Jen, T.-Y., Laurent, D., Spyratos, N., & Sy, O. (2006). Towards mining frequent queries in star schemes. In *Proceedings of the 4th International Workshop (KDID 2005), Porto, Portugal, October 3, 2005* (pp. 104–123).: Springer.

[Johnson & Klug, 1984] Johnson, D. S. & Klug, A. C. (1984). Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences (JCSS)*, 28(1), 167–189.

[Kamber et al., 1997] Kamber, M., Han, J., & Chiang, J. (1997). Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997* (pp. 207–210).

[Kantola et al., 1992] Kantola, M., Mannila, H., Räihä, K. J., & Siirtola., H. (1992). Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7, 591–607.

[Kivinen & Mannila, 1995] Kivinen, J. & Mannila, H. (1995). Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1), 129–149.

[Koeller & Rundensteiner, 2006] Koeller, A. & Rundensteiner, E. (2006). Heuristic strategies for the discovery of inclusion dependencies and other patterns. *Journal on Data Semantics V*, (pp. 185–210).

[Koeller & Rundensteiner, 2003] Koeller, A. & Rundensteiner, E. A. (2003). Discovery of high-dimensional inclusion dependencies. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India* (pp. 683–685). Los Alamitos, CA, USA: IEEE Computer Society.

[Koopman & Siebes, 2008] Koopman, A. & Siebes, A. (2008). Discovering relational item sets efficiently. In *Proceedings of the SIAM International Conference on Data Mining (SDM), April 24-26, 2008, Atlanta, Georgia, USA* (pp. 108–119).

[Koopman & Siebes, 2009] Koopman, A. & Siebes, A. (2009). Characteristic relational patterns. In J. F. E. IV, F. Fogelman-Soulié, P. Flach, & M. Zaki (Eds.), *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009* (pp. 437–446).: ACM.

[Kuramochi & Karypis, 2001] Kuramochi, M. & Karypis, G. (2001). Frequent subgraph discovery. In N. Cercone, T. Lin, & X. Wu (Eds.), *Proceedings of* *the 2001 IEEE International Conference on Data Mining (ICDM 2001)* (pp. 313–320).: IEEE Computer Society Press.

[Liu et al., 2002] Liu, H., Hussain, F., Tan, C. L., & Dash, M. (2002). Discretization: An enabling technique. *Data Mining and Knowledge Discovery*, 6(4), 393–423.

[Lopes et al., 2000] Lopes, S., Petit, J.-M., & Lakhal, L. (2000). Efficient discovery of functional dependencies and armstrong relations. In C. Zaniolo, P. C. Lockemann, M. H. Scholl, & T. Grust (Eds.), *Proceedings of the 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000*, volume 1777 of *Lecture Notes in Computer Science* (pp. 350–364).: Springer.

[Lopes et al., 2002] Lopes, S., Petit, J.-M., & Lakhal, L. (2002). Functional and approximate dependency mining: database and FCA points of view. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2-3), 93–114.

[Mannila & Toivonen, 1997] Mannila, H. & Toivonen, H. (1997). Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3), 241–258.

[Mannila et al., 1994] Mannila, H., Toivonen, H., & Verkamo, A. I. (1994). Efficient algorithms for discovering association rules. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases* (pp. 181–192).: AAAI Press.

[Marchi et al., 2004] Marchi, F. D., Flouvat, F., & Petit, J.-M. (2004). Adaptive strategies for mining the positive border of interesting patterns: Application to inclusion dependencies in databases. In J.-F. Boulicaut, L. D. Raedt, & H. Mannila (Eds.), *Proceedings of Constraint-Based Mining and Inductive Databases, European Workshop on Inductive Databases and Constraint Based Mining, Hinterzarten, Germany, March 11-13, 2004*, volume 3848 of *Lecture Notes in Computer Science* (pp. 81–101).: Springer.

[Marchi et al., 2002] Marchi, F. D., Lopes, S., & Petit, J.-M. (2002). Efficient algorithms for mining inclusion dependencies. In C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, & M. Jarke (Eds.), *Proceedings of the 8th International Conference on Extending Database Technology (EDBT 2002), Prague, Czech Republic, March 25-27*, volume 2287 of *Lecture Notes in Computer Science* (pp. 464–476).: Springer.

[Marchi & Petit, 2003] Marchi, F. D. & Petit, J.-M. (2003). 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 (pp. 27–34).: IEEE Computer Society.

[Matos & Grasser, 2004] Matos, V. & Grasser, B. (2004). SQL-based discovery of exact and approximate functional dependencies. *ACM SIGCSE Bulletin*, 36(4), 58–63.

[Ng et al., 2002] Ng, E., Fu, A., & Wang, K. (2002). Mining association rules from stars. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*, volume 20 (pp. 30–39).

[Nijssen & Kok, 2003a] Nijssen, S. & Kok, J. N. (2003a). Efficient frequent query discovery in FARMER. In N. Lavrac, D. Gamberger, H. Blockeel, & L. Todorovski (Eds.), *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases, Cavtat-Dubrovnik, Croatia, September 22-26, 2003*, volume 2838 of *Lecture Notes in Computer Science* (pp. 350–362).: Springer.

[Nijssen & Kok, 2003b] Nijssen, S. & Kok, J. N. (2003b). Proper refinement of datalog clauses using primary keys. In *Proccedings of the 2003 Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2003).*

[Novelli & Cicchetti, 2001] Novelli, N. & Cicchetti, R. (2001). FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)* (pp. 189–203). London, UK: Springer-Verlag.

[Pasquier et al., 1999] Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In C. Beeri & P. Buneman (Eds.), *Proceedings of the 7th International Conference on Database Theory (ICDT '99), Jerusalem, Israel, January 10-12, 1999*, volume 1540 of *Lecture Notes in Computer Science* (pp. 398–416).: Springer.

[Pei et al., 2000] Pei, J., Han, J., & Mao, R. (2000). CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (pp. 21–30).

[Piatetsky-Shapiro, 1991] Piatetsky-Shapiro, G. (1991). Discovery, analysis, and presentation of strong rules. In *Knowledge Discovery in Databases* (pp. 229–248). AAAI/MIT Press.

[Plotkin, 1970] Plotkin, G. (1970). A note on inductive generalization. In *Machine Intelligence*, volume 5 (pp. 153–163). Edinburgh University Press.

[Sánchez et al., 2008] Sánchez, D., Serrano, J.-M., Blanco, I., Martín-Bautista, M. J., & Miranda, M. A. V. (2008). Using association rules to mine for strong approximate dependencies. *Data Mining and Knowledge Discovery*, 16(3), 313–348.

[Siebes et al., 2006] Siebes, A., Vreeken, J., & van Leeuwen, M. (2006). Item sets that compress. In J. Ghosh, D. Lambert, D. B. Skillicorn, & J. Srivastava (Eds.), *Proceedings of the SIAM International Conference on Data Mining 2006 (SDM)* (pp. 393—404).: SIAM.

[Srikant & Agrawal, 1996] Srikant, R. & Agrawal, R. (1996). Mining quantitative association rules in large relational tables. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (pp. 1–12).: ACM New York, NY, USA.

[Tsechansky et al., 1999] Tsechansky, M. S., Pliskin, N., Rabinowitz, G., & Porath, A. (1999). Mining relational patterns from multiple relational tables. *Decision Support Systems*, 27(1-2), 177–195.

[Ullman, 1988] Ullman, J. (1988). *Principles of database and knowledge-base systems, volume 1*, volume 1 of *Principles of Computer Science*. Computer Science Press.

[Ullman, 1989] Ullman, J. (1989). *Principles of database and knowledge-base systems, volume 2*, volume 14 of *Principles of Computer Science*. Computer Science Press.

[Webb, 2000] Webb, Geoffrey, I. (2000). Efficient search for association rules. In *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '00)* (pp. 99–107). New York, NY, USA: ACM.

[Weisstein, 2009] Weisstein, E. W. (2009). Restricted growth string. From MathWorld – A Wolfram Web Resource.

[Wyss et al., 2001] Wyss, C. M., Giannella, C., & Robertson, E. L. (2001). FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In Y. Kambayashi, W. Winiwarter, & M. Arikawa (Eds.), *Proceedings of the 3rd International Conference Data Warehousing and Knowledge Discovery (DaWaK 2001), Munich, Germany, September 5-7, 2001*, volume 2114 of *Lecture Notes in Computer Science* (pp. 101–110).: Springer.

[Yan & Han, 2002] Yan, X. & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)* (pp. 721).

[Yannakakis, 1981] Yannakakis, M. (1981). Algorithms for acyclic database schemes. In *Proceedings of the seventh international conference on Very Large Data Bases (VLDB '1981)* (pp. 82–94).: VLDB Endowment.

[Yao & Hamilton, 2008] Yao, H. & Hamilton, H. J. (2008). Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2), 197–219.

[Zaki, 2000] Zaki, M. J. (2000). Generating non-redundant association rules. In *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '00)* (pp. 34–43). New York, NY, USA: ACM.

[Zaki, 2002] Zaki, M. J. (2002). Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '02)* (pp. 71–80). New York, NY, USA: ACM.

[Zaki, 2004] Zaki, M. J. (2004). Mining non-redundant association rules. *Data Mining and Knowledge Discovery*, 9(3), 223–248.

[Zaki & Gouda, 2003] Zaki, M. J. & Gouda, K. (2003). Fast vertical mining using diffsets. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '03)* (pp. 326–335). New York, NY, USA: ACM.

[Zaki & Hsiao, 2002] Zaki, M. J. & Hsiao, C.-J. (2002). CHARM: An efficient algorithm for closed itemset mining. In R. L. Grossman, J. Han, V. Kumar, H. Mannila, & R. Motwani (Eds.), *Proceedings of the 2nd SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*: SIAM.

[Zaki et al., 1997] Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. In *Proceedings of the 3rd Internationl Conference on Knowledge Discovery and Data Mining (KDD)* (pp. 283–286).