Characteristic Relational Patterns

Arne Koopman
Department of Computer Science
Universiteit Utrecht
koopman@cs.uu.nl

Arno Siebes
Department of Computer Science
Universiteit Utrecht
arno@cs.uu.nl

ABSTRACT

Research in relational data mining has two major directions: finding global models of a relational database and the discovery of local relational patterns within a database. While relational patterns show how attribute values co-occur in detail, their huge numbers hamper their usage in data analysis. Global models, on the other hand, only provide a summary of how different tables and their attributes relate to each other, lacking detail of what is going on at the local level.

In this paper we introduce a new approach that combines the positive properties of both directions: it provides a detailed description of the complete database using a small set of patterns. More in particular, we utilise a rich pattern language and show how a database can be encoded by such patterns. Then, based on the MDL-principle, the novel RDB-KRIMP algorithm selects the set of patterns that allows for the most succinct encoding of the database. This set, the code table, is a compact description of the database in terms of local relational patterns. We show that this resulting set is very small, both in terms of database size and in number of its local relational patterns: a reduction of up to 4 orders of magnitude is attained.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

Keywords

Frequent Patterns, Relational Data Mining, MDL

1. INTRODUCTION

Relational data mining seeks to generalise traditional, single-table data analysis to the analysis of multiple inter-related tables. As relational models are more expressive than their single table counterparts, they allow for a more succinct description of the database than when one has to summarize each and every table and all their connections separately. Current research in relational data mining follows one of two directions. Either one aims to find a *'global'* model of the complete database, or one seeks interesting *'local'* patterns in the database.

Both directions have their specific merits and shortcomings. A *'global'* database model, for example a Probabilistic Relational Model (PRM) [9], is often small and interpretable, but it lacks detail as it only shows how the attributes of the tables interact. On the other hand, one can use an approach like WARMR to find frequent patterns [1, 3]. These patterns can be regarded as *'local'* models that describe a partial structure of the relational database. A well-known drawback of this method is the exponential pattern set growth when mining for less frequent, but often interesting, patterns.

In this paper we present a new approach, RDB-KRIMP, that combines the strengths of both approaches. It finds a global model that describes the complete database using only a small set of *characteristic* relational patterns. While collectively the patterns show the global structure, individually they reveal the local interactions in the database. A global model that describes the complete relational database must capture the behaviour of all tables and their interactions, and thus requires a focus on all tables. While frequent pattern mining techniques prune the pattern search space using a *single* 'target' table, *all* tables function as 'targets' in our global approach. Hence, our pattern language is essentially that of FARMER [13] except we do not restrict the patterns to have their 'roots' in a single target table.

Given this pattern set, we use the Minimum Description Length (MDL) principle [7] to select a small set of *characteristic* patterns for the database. RDB-KRIMP uses the MDL-principle to find those patterns that together describe the database well. In contrast to KRIMP [14], this enriched pattern language allows RDB-KRIMP to find more complex patterns in the database. However, finding these richer models for *relational* databases requires a novel lossless encoding scheme that relies on its ordering. During the encoding, RDB-KRIMP reorders the tuples based on the patterns of the model such that the encoded database can always be decoded in a lossless fashion. RDB-KRIMP proves effective: with its lossless encoding, it can find models that stay compact and utilise our enhanced pattern language.

After presenting our theoretical framework and the RDB-KRIMP algorithm, we validate our claims with experimental results obtained on publicly available KDDcup databases. We show that the global models stay compact, in contrast to the original candidate pattern sets that grow large for lower minimum supports.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'09, June 28–July 1, 2009, Paris, France.
Copyright 2009 ACM 978-1-60558-495-9/09/06 ...$5.00.

Furthermore, we show that we attain good results as a result of the specific characteristics of our enhanced pattern language. We show that simpler pattern sets lead to worse results and that target table based approaches are much less effective. We conclude by discussing the readability of our models, and their related work.

2. DATA AND PATTERNS

In this section we formally define the data type, relational patterns, how such patterns occur in the database, and how to calculate the support of a pattern.

2.1 Data

We assume that the data resides in a multi-relational database in which the relations between the tuples in the various tables is coded, as usual, via *foreign keys*. We assume that any pair of two tables have at most one foreign key relationship between them. This is without loss of generality as databases can always be losslessly recoded such that this assumption holds. Moreover, we assume that all attributes of all tables have a categorical domain. To introduce our notation, we give a brief formal description.

The database db consists of a set of tables, $db = \{T^1, \ldots, T^n\}$, and we assume that all the table names (the T^i) are unique. Each table T has a schema $S(T)$. This schema consists of a *key*, 0 or more *foreign keys*, and 1 or more attributes, i.e,

$$S(T^i) = (K^i, \mathcal{F}^i, \mathcal{A}^i)$$

in which:

- K^i is the key. Without loss of generality, we assume that the key name is unique. Its domain, $D(K^i)$, is a set of integers.
- $\mathcal{F}^i$ is a set of 0 or more foreign keys. We assume that the database schema is consistent, that is
  - For each foreign key F^i_j of T^i, there is a table, say, $T^l \in db$ for which it is the key, i.e., $F^i_j = K^l$. As noted above, we assume that there is at most one foreign key in T^i that refers to K^l
  - The domain of F^i_j is the domain of that K^l.
- $\mathcal{A}^i$ consists of 1 or more attributes. Each attribute A^i_k has a categorical domain $D(A^i_k)$.
- Summing up, the domain of table T^i denoted by $D(T^i)$ is the cartesian product of all domains involved, i.e.,

$$D(T^i) = D(K^i) \times \prod_{F^i_j \in \mathcal{F}^i} D(F^i_j) \times \prod_{A^i_k \in \mathcal{A}^i} D(A^i_k)$$

In our example database, shown bottom-right in Figure 1, the DISPOSITION table has as key: $disp_{ID}$ (depicted in bold) and as foreign key $account_{ID}$. Moreover, it has an attribute *Type* whose domain is: $\{Owner, Disponent\}$.

Next to a schema, each table has an *extend*, consisting of a set of tuples. As usual, we blur the distinction between the table and its extend and say that a tuple t is in table T^i; denoted by $t \in T^i$. The database as whole should satisfy *referential integrity*. That is, foreign-key values in a tuple refer to existing tuples in the table for which this foreign key is the key. More formally we have:

A tuple for table T^i with $S(T^i) = (K^i, \mathcal{F}^i, \mathcal{A}^i)$ is given by:

$$t = (K^i = k, \{F^i_j = k_j\}, \{A^i_l = v_l\})$$

in which:

T^i

K^i	F^i_j	A^i_1	A^i_2
k	k_j	v_1	v_2

T^1 = ACCOUNT

$account_{ID}$	Frequency	Date
10	2	06/2007
11	3	03/2006
12	3	08/2006
13	2	03/2006
14	1	05/2008

T^2 = LOAN

$loan_{ID}$	$account_{ID}$	Date	Amount	Duration	Payment
30	10	06/2008	10245	12	A
31	10	09/2008	13722	24	B
32	11	08/2006	27313	36	B
33	12	09/2006	27147	12	B
34	12	05/2008	27194	36	D
35	13	09/2008	30289	12	B
36	13	06/2008	18203	12	C

T^3 = ORDER

$order_{ID}$	$account_{ID}$	Bank-To	Amount-To	Amount	Type
20	10	ST	141	1000	UVER
21	10	QR	359	2000	SIPO
22	11	YZ	850	1000	SIPO
23	13	ST	283	1000	NULL
24	13	OP	850	2000	SIPO

T^4 = DISPOSITION

$disp_{ID}$	$account_{ID}$	Type
40	10	OWNER
41	11	DISPONENT
42	11	OWNER
43	12	DISPONENT
44	12	OWNER

Figure 1: An illustrative relational database: an excerpt from the Financial database.

- $k \in D(K^i)$,
- the tuple has one entry for each $F^i_j \in \mathcal{F}^i$ and one entry for each $A^i_j \in \mathcal{A}^i$.
- let K_l be the key to which F^i_j refers, then $k_j \in D(K^l)$,
- for referential integrity on the database db, we have that for any tuple $t \in T^i$, there is a tuple $t' \in T^l$ such that $\pi_{F^i_l}(t) = \pi_{K^l}(t')$.
- $v_l \in D(A^i_l)$.

Again as usual, we will suppress the labels in tuples whenever possible. That is, we simply write $(40, 10, Owner) \in$ DISPOSITION for a tuple in our example database in Figure 1.

2.2 Patterns

The prototypical example of patterns are item sets. In the case of a (single) table of categorical data, an item set generalises to a *selection*. Clearly such patterns should be included into our pattern language. However, "true" relational patterns should cross multiple tables. That is, they should describe related selections over multiple tables.

After formally introducing our pattern definition, we will illustrate it with an example.

DEFINITION 1 (PATTERN). *Let $db = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{F}^i, \mathcal{A}^i)$.*

- *Let $\{A_1, \ldots, A_l\} \subseteq \mathcal{A}^i$ and let $v_j \in D(A_j)$, then the expression p defined as*

$$p = T^i(\{A_1 = v_1, \ldots, A_l = v_l\})$$

is a pattern *for T^i; this is denoted by $p \in \mathcal{P}^i$.*

- *Let T^i have key K^i, moreover, let K^i be a foreign key of T^j; that is, there is an $F^j_l \in \mathcal{F}^j$ such that $F^j_l = K^i$. Let $p_0 \in \mathcal{P}^i$ and let $\{p_1, \ldots p_k\} \subseteq \mathcal{P}^j$. The expression p defined as*

$$p = p_0[p_1, \ldots, p_k]$$

is a pattern for T^i, i.e., $p \in \mathcal{P}^i$.

- *Let T^i have key K^i, moreover, let K^i be a foreign key of the q tables $T^{j_1}, \ldots, T^{j_q}$, such that if $r \neq s$, then $T^{j_r} \neq T^{j_s}$. Let $p_0 \in \mathcal{P}^i$ and for $l \in \{1, \ldots, q\}$, let $\{p^l_1, \ldots p^l_{k_l}\} \subseteq \mathcal{P}^{j_l}$. The expression p defined as*

$$p = p_0[[p^1_1, \ldots, p^1_{k_1}], \ldots, [p^q_1, \ldots p^q_{k_q}]]$$

is a pattern for T^i, i.e., $p \in \mathcal{P}^i$.

The third component is a generalisation of the second, which is included to simplify part of the following definitions.

In our pattern definition we define the *alphabet patterns* as those patterns that select one attribute and assign it to one value (i.e. $p = T^i(A_j = v_j)$).

Also, we define the $size$ of a pattern as the number of attributes within the pattern:

$$size(p = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})) = j$$
$$size(p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^m, \ldots p_{k_m}^m]]) =$$
$$size(p_0) + \sum_{i=1}^{m} \sum_{j=1}^{k_i} size(p_j^i).$$

Using our example database shown in Figure 1, we can now illustrate our pattern definition. An example of the first component, a single table selection on the ACCOUNT table, would be ACCOUNT({Frequency =2}). This pattern can be seen in the database in the set of tuples with account-ids 10 and 13.

The second component allows a pattern to have multiple selections from a table. As an example, in our database we see the pattern that *accounts of Frequency 2* have *dispositions of type Owner* (seen at account-id 10). A more complex example is that there are *accounts of Frequency 3* that have both *dispositions of type Owner* and *dispositions of type Disponent*. For this last pattern, one tuple in the ACCOUNT table is related to two distinct tuples in the DISPOSITION table (account-ids 11 and 12). This last pattern is represented as follows: ACCOUNT({Frequency = 3}) [DISPOSITION({Type = Owner}), DISPOSITION({Type = Disponent})].

The third component extends this last example by allowing the patterns to span more than two tables.

Note that in our pattern language a pattern can 'start' at one of the tables $T \in \{T^1, \ldots, T^n\}$. We do not restrict the 'root' of these patterns to be at one specific target table, because interesting patterns within the database can start from all possible tables. Furthermore, as one single tuple can be joined with multiple other tuples, the structure of our patterns matches the complexity of the database.

Having a richer pattern language is no virtue in its own right. Alternatively, we could have used pattern languages such as those used in WARMR [3], FARMER [13], or in our earlier work [10]. The experiments however show that our pattern set, consisting of FARMER pattern sets without a fixed target table, allows us to capture more important structure within the data.

2.3 Pattern Occurrences

Our the patterns can become rather complicated structures, containing lists of lists of lists of ... of tuples. Hence, it is illustrative to consider what the domain of such patterns is. That is, what does an instance look like? The definition of these domains follows the inductive structure of the patterns.

DEFINITION 2 (DOMAIN). *Let $db = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{F}^i, \mathcal{A}^i)$. Moreover, let $p \in \mathcal{P}^i$. The domain of p, denoted by $D(p)$, is given by*

- *If $p = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})$, then $D(p) = D(T^i)$.*
- *If $p = p_0[p_1, \ldots, p_k]$, then*
$$D(p) = D(p_0) \times [D(p_1), \ldots, D(p_k)].$$
- *If $p = p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^q, \ldots p_{k_q}^q]]$, then D(p) =*
$$D(p_0) \times [[D(p_1^1), \ldots, D(p_{k_1}^1)], \ldots, [D(p_1^q), \ldots D(p_{k_q}^q)]].$$

Algorithm 1 Generate P_θ

Generate $P_\theta(db, \theta)$
1: **for all** $T^i \in db$ **do**
2: $\quad P_\theta^i = FARMER(db, \theta, target = T^i)$
3: **end for**
4: $P_\theta = \bigcup_i P_\theta^i$
5: **return** P_θ

Note that this domain definition is broad: it does not enforce that patterns describe related tuples. The reason for this liberal definition is that here we are only interested in the general structure of the domain. Referential integrity does play its role in the definition of an occurrence of a pattern.

While these domains may have a rather complicated structure, there is some simplicity. For a pattern $p \in \mathcal{P}^i$, the domain is either $D(T^i)$ or it is the cartesian product of $D(T^i)$ with a complicated list domain. That is, $D(p) = D(T^i) \times X$, in which X denotes a list domain. This observation has a useful consequence. It means that if p is a pattern for T^i (i.e., $p \in \mathcal{P}^i$), and t is an *instance* of p (i.e, $t \in D(p)$), we can project t on the keys, the foreign keys and the attributes of T^i. We will use these projections in the definition of an occurrence.

An occurrence of a pattern in the database is an instance of that pattern in the database. However, different from these instances, for occurrences we do require *referential integrity*. That is, the occurrences should consist of related tuples only. This makes the definition slightly more complex.

DEFINITION 3 (OCCURRENCE). *Let $db = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{F}^i, \mathcal{A}^i)$. Moreover, let $p \in \mathcal{P}^i$.*

- *If $p = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})$, occ(p) is the set of those tuples $t \in T^i$ for which*
$$\forall k \in \{1, \ldots, j\} : \pi_{A_k}(t) = v_k$$
- *If $p = p_0[p_1, \ldots, p_k]$, we know that $p_0 \in \mathcal{P}^i$ and that there is a table T^j such that firstly $\{p_1, \ldots, p_k\} \subseteq \mathcal{P}^j$ and secondly that the key K^i of T^i is a foreign key, say F_l^j of T^j. An instance $t = (t_0, [t_1, \ldots, t_k])$ of p is an occurrence of p, denoted by $t \in occ(p)$, if:*
  - *$t_0 \in occ(p_0)$ and for $m \in \{1, \ldots, k\} : t_m \in occ(p_m)$.*
  - *for $m, n \in \{1, \ldots, k\} : m \neq n \rightarrow t_m \neq t_n$*
  - *for $m \in \{1, \ldots, k\} : \pi_{F_l^j}(t_m) = \pi_{K^i}(t_0)$*
- *If $p = p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^q, \ldots p_{k_q}^q]]$, then an instance $t \in D(p)$ given by*
$$t = (t_0, [[t_1^1, \ldots, t_{k_1}^1], \ldots, [t_1^q, \ldots, t_{k_q}^q]])$$
is an occurrence of p (i.e., $t \in occ(p)$), if
$$\forall l \in \{1, \ldots, q\} : (t_0, [t_1^l, \ldots, t_{k_l}^l]) \in occ(p_0[p_1^l, \ldots, p_{k_l}^l]).$$

As usual, the *support* of a pattern is the number of its occurrences. We say that a pattern is frequent if the support exceeds some user-defined threshold called the minimum support θ.

Note that we use lists in our occurrence definition. Each occurrence can span multiple tuples within one table T^i that are stored in a list $[t_1^i, \ldots, t_{k_i}^i]$. We will preserve this order of the tuples as it

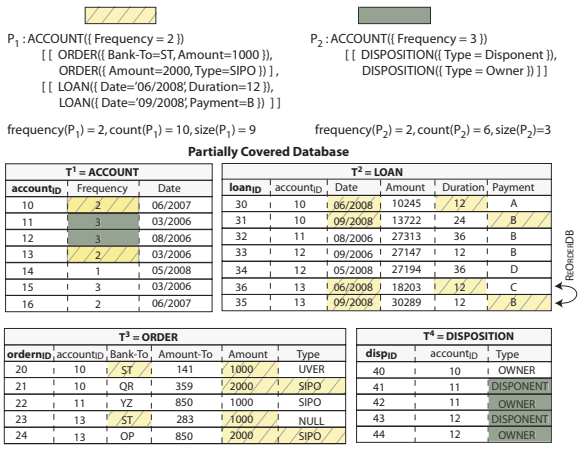

Figure 2: The database is partially covered with the two first patterns of the code table using RDB-KRIMP**. The uncoloured part of the database is covered by alphabet patterns. Note that for a lossless decoding we incorporate the database order (seen at the swap of** LOAN**:loan-id=35 and 36).**

is essential to encode the database in a lossless manner, as we will show below.

Given this pattern definition, in order to derive the set of frequent patterns P_θ one can resort to existing relational mining algorithms like FARMER [13] or attribute tree miners like FATminer [2]. Partial frequent pattern sets generated by either approach can be combined into P_θ (see Algorithm 1).

Finally, we define a canonical order on our patterns. We assume for each table T^i, each attribute A_j, and each attribute value v_k a unique (string) label. We denote by $l(X)$ the unique label assigned to X ($X \in \{T^i, A_j, v_k\}$). Canonical forms are simply strings, hence we have the familiar lexicographic order, denote by $<_{lex}$, on them. Using this order, we define:

$$canonical(p = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})) = l(T^i) : \{l(A_1) : l(v_1), \ldots, l(A_j) : l(v_j)\}$$

$$canonical(p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^m, \ldots p_{k_m}^m]]) = canonical(p_0) :_{i=1}^{m} :_{j=1}^{k_i} canonical(p_j^i)$$

This allows us to define a canonical order on our patterns: $p_0 <_{can} p_1$ iff $canonical(p_0) <_{lex} canonical(p_1)$.

3. PROBLEM STATEMENT

Now we have defined our patterns and database, we can present our problem formally. In order to find a good global model for our relational database, we use the minimum description length (MDL) [7] principle, which is a practical application of Kolmogorov Complexity [11]. Given a set of models $\mathcal{H}$, we want to find a model H that minimises $L(H) + L(D|H)$, in which

- $L(H)$ is the length, in bits, of the description of H, and
- $L(D|H)$ is the length, in bits, of the description of the data D when encoded with H.

The data that is encoded by our model resides within a relational database as defined in Section 2, or more specifically within its attribute data. Similar to [10, 14] our models are code tables. Such

Algorithm 2 REORDERDB

REORDERDB $(reorder, [t_1, \ldots, t_i])$
1: **if** $[t_1, \ldots, t_i] \cap reorder \subseteq^{ord} reorder$ **then**
2: $\quad reorder = reorder \cup [t_1, \ldots, t_i]$
3: $\quad$ **return true**
4: **else**
5: $\quad$ **return false**
6: **end if**

$A : [a_1, \ldots, a_n] \subseteq^{ord} B : [b_1, \ldots, b_m]$
1: **if** $A = \varnothing$ **then**
2: $\quad$ **return true**
3: **else if** $B = \varnothing$ **then**
4: $\quad$ **return false**
5: **else if** $a_1 = b_1$ **then**
6: $\quad$ **return** $[a_2, \ldots, a_n] \subseteq^{ord} [b_2, \ldots, b_m]$
7: **else if** $a_1 \neq b_1$ **then**
8: $\quad$ **return** $[a_1, \ldots, a_n] \subseteq^{ord} [b_2, \ldots, b_m]$
9: **end if**

a code table CT is a two column table. On the left hand side reside relational patterns as defined above, on the right hand side reside the codes. The codes are taken from a prefix code $\mathcal{C}$. For each code, we calculate a Shannon entropy based length: more frequently used codes obtain smaller lengths. Further, the number of patterns in the code table is denoted by $|CT|$.

We encode the relational database using the patterns from a code table CT. Figure 2 shows an example on how this encoding comes about through a database cover. Here, we show two patterns that (partially) cover the database. Using our pattern notation, we write the first code table pattern as:

ACCOUNT({Frequency = 2})[
[ORDER({Bank-To = ST, Amount = 1000}),
ORDER({Amount = 2000, Type = SIPO})]
[LOAN({Date = 06/2008, Duration=12}),
LOAN({Date = 09/2008, Payment=B})]]

We cover the database by replacing the related attribute values with the code of the pattern. As this pattern occurs at two yet uncovered locations in the database (id=10 and 13), it is used to describe this part of the database. Note that in this figure each code table element has its own distinct colour.

In order for the encoding to be lossless, we need to be able to decode every part of the occurrence. As an occurrence may span multiple tuples within the database, we need to write the code at each tuple covered by this occurrence. Furthermore, as each pattern has just one code the necessity of a database- and pattern order becomes clear: we need to know which tuple is covered with which pattern from the list.

We match the order of the tuples within the database with the order of the pattern. In the first pattern (p_1 in fig. (2)), LOAN : $\{Date = 06/2008, Duration = 12\}$ is ordered before LOAN : $\{Date = 09/2008, Payment = B\}$. Note the swap of tuples 35 and 36 to align the database order with the order of the pattern. Once covered, the complete database is encoded and looks like a mosaic, which can be decoded using the database order, and the code table (see fig. 2).

So, for unambiguous decoding, the order of the tuples in the database has to be aligned with the order in the code table patterns. Therefore, we allow a (partial) re-ordering of the tuples in the database. However, a pair of tuples is only re-ordered once,

Algorithm 3 COVER and COVERDB

COVER $(db, p, reorder)$

1: $frequency = 0$
2: **for all** $\{t\} \in occ(p)$ **do**
3: **if** $p \subseteq \{A^t\}$ **then**
4: **if** REORDERDB$(reorder, \{t\})$ **then**
5: $\{A^t\} = \{A^t\} \setminus p$
6: $frequency$++
7: **end if**
8: **end if**
9: **end for**
10: **return** $frequency$

COVERDB (db, CT)

1: $reorder = \varnothing$
2: **for all** $c_i \in CT$ **do**
3: $frequency(c_i) =$ COVER(db, $c_i, reorder$)
4: $count(c_i) = frequency(c_i) \times |coverspots(c_i)|$
5: **end for**

otherwise the unambiguous decoding property will be lost. Hence, we keep track of the order. Initially this ordered list, $reorder$, is empty. Whenever two (or more tuples) that are not yet in the list are re-ordered their identifiers are appended.

We cover the database with patterns p until the database is completely covered. An occurrence of a pattern can only cover the database if the database order can be aligned to match the pattern. This is the case when the tuples related to the occurrence are not yet ordered, or if they are already ordered in the correct order (e.g. the order of the tuples matches the order of the pattern). We update and check partial database order via the REORDERDB algorithm (see Algorithm 2). The algorithm takes as input the current (partial) database order $reorder$ and a list of tuples of the current occurrence $[t_1, \ldots, t_i]$. Initially the database order $reorder$ is an empty list as the database is unordered. To check whether $[t_1, \ldots, t_i] \cap reorder$ is an ordered subset of $reorder$ we use the $\subseteq^{ord}$ operator (line 1). If so, the order is updated by appending the current list of tuples that are not yet part of the database order (2). Otherwise, this particular occurrence cannot be used to cover the database.

Now that we can adjust $reorder$ to align with a pattern occurrence, we can partially cover a database given a single code table pattern (see Algorithm 3). For each pattern, we have a set of occurrences $occ(p)$ (1). COVER iterates over all occurrences and evaluates whether it can be covered (3). We only cover the current occurrence (the list of tuples) if all related attributes are still uncovered (4), and if the current occurrence is an ordered subset of the current database order (5). If so, the attributes of the occurrence are covered and the frequencies are updated (6,7). We define the *frequency* of a pattern as the number of times we cover the database with it.

Using the code table CT, we cover the complete relational database using the COVERDB algorithm (see Algorithm 3). Considering the patterns in the code table, it covers the database using the COVER algorithm (2-3). During the cover process, we obtain for each code table pattern its $frequency$ (line 3).

To be able to decode the encoded database, we have to write its code at each involved tuple. We define $coverspots$ as the set of tuples at which we need to write down the code for pattern p. More

Algorithm 4 RDB-KRIMP

RDB-KRIMP (db, P_θ)

1: $CT = CT_{init}$
2: **for all** $c \in P_\theta$ **do**
3: $CT_{new} = CT + c$ in order
4: COVERDB(db, CT_{new})
5: **if** $\mathcal{L}(CT_{new}, db) < \mathcal{L}(CT, db)$ **then**
6: $CT = CT_{new}$
7: **end if**
8: **end for**

formally, we define $coverspots$ as:

$$coverspots(p = T^i(\{A_i = v_i\})) = \{t\}$$
$$coverspots(p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^m, \ldots p_{k_m}^m]]) = coverspots(p_0) \cup_{i=1}^{m} \cup_{j=1}^{k_i} coverspots(p_j^i)$$

We denote the number of $coverspots$ by: $|coverspots(c)|$.

Consider the first pattern, p_1 , from Figure 2. The number of $coverspots$ is 5, as each occurrence has five distinct tuples in the database associated with it (one in ACCOUNT, two in ORDER, and two in LOAN). We denote the $count$ as the total number of times we have to write the code given a code table pattern c : $count(c) = frequency(c) \times |coverspots(c)|$ (line 4).

Given the obtained counts, we can now determine the code length for each code table element c_i. For optimal encoding, we use a Shannon code, i.e.,

$$\mathcal{L}_{CT}(c_i) = -\log(\frac{count(c_i)}{\sum_{c_j \in CT} count(c_j)}).$$

Each code table element c has a standard length, which is the decoded length using alphabet patterns only: $\mathcal{L}_{st}(c)$. The encoded length of the complete code table then is:

$$\mathcal{L}(CT) = \sum_{c_i \in CT} \mathcal{L}_{CT}(c_i) + \mathcal{L}_{st}(c_i).$$

In covering the database, we write $count(c_i)$ codes for each code table element. Given the code lengths, the encoded size of the database becomes

$$\mathcal{L}_{CT}(db) = \sum_{c_i \in CT} \mathcal{L}_{CT}(c_i) \times count(c_i).$$

In our approach, we define the total encoded size as:

$$\mathcal{L}(db, CT) = \mathcal{L}(CT) + \mathcal{L}_{CT}(db).$$

Now that we can derive the total encoded size of the database given a code table, we can formulate our problem as follows.

PROBLEM STATEMENT. *Let db* $= \{T^1, \ldots T^n\}$ *be a relational database as defined in Section 2. Find the code table* CT *that minimises* $\mathcal{L}(CT, db)$.

4. RDB-KRIMP ALGORITHM

Finding the optimal code table, the one that compresses the database best, is a very hard problem. The search space is extremely large and there is no useful structure to prune it. Hence, we have to use heuristics and we therefore resort to an approach similar to the one described in [14].

The algorithm we introduce to this end, RDB-KRIMP, approximates the optimal code table for a given database. It does this by starting with the simplest possible code table and iteratively testing

Table 1: Characteristics of the used databases. Shown are for each table, the number of tuples ($\#t$), the number of attributes ($\#a$), the number of keys (K and $\mathcal{F}$) ($\#k$), and the average number of joins a single tuple can make ($\overline{join}$).

	table	$\#t$	$\#a$	$\#k$	$\overline{join}$
FINANCIAL	ACCOUNT	682	2	3	5.43
	CLIENT	827	2	3	2
	LOAN	682	5	2	6.70
	CARD	36	2	3	1
	DISP	827	1	4	1.04
	ORDER	1513	3	3	1
GENES	GENES1	862	4	1	6.86
	GENES2	862	4	1	6.86
	INT	910	2	3	1
	META1	4151	4	2	1
	META2	4151	4	2	1
HEPATITIS	BIO	694	5	2	1
	IFN	198	4	2	1
	OLAB	31039	3	2	1
	PATIENT	771	3	1	42.4

each pattern in a candidate set. A candidate pattern is only kept in the code table if it improves the compression. The RDB-KRIMP algorithm is shown in pseudo code in Algorithm 4.

RDB-KRIMP starts with a database db and a frequent pattern set P_θ as input. To ensure that the complete database can be covered always, the code table is initialized with CT_{init} (line 1), which contains all alphabet elements (i.e. $p = T^i(A_j = v_j)$). One by one, it takes a candidate pattern from P_θ and tests whether it contributes to improve compression (2-8). To do this, a new code table CT_{new} is constructed by adding the candidate pattern to the previous code table CT (3). Using this code table we compute a cover of the database (4) and the compressed sizes of the old and new code table are compared (5). If the addition of the new candidate pattern improves compression, it is kept in the code table (6). Otherwise, it is permanently discarded.

Note that we need to define two orders: the first on the candidate pattern set P_θ and the second on the patterns in the code table. For P_θ we define an order for all pattern pairs (p_1, p_2):

if $support(p_1) > support(p_2) \rightarrow p_1 > p_2$
  else if $size(p_1) > size(p_2) \rightarrow p_1 > p_2$
    else if $p_1 >_{can} p_2 \rightarrow p_1 > p_2$
      else $p_1 \leq p_2$

For the code table, we define the following order on all pattern pairs (p_1, p_2):

if $size(p_1) > size(p_2) \rightarrow p_1 > p_2$
  else if $support(p_1) > support(p_2) \rightarrow p_1 > p_2$
    else if $p_1 >_{can} p_2 \rightarrow p_1 > p_2$
      else $p_1 \leq p_2$

We assume that both the frequent pattern sets and code tables are always ordered in this fashion. Note that we are not after the actual attained compression, but rather the patterns that contribute to it. We use lossless compression as a means, not as a goal, to find a good set of descriptive patterns.

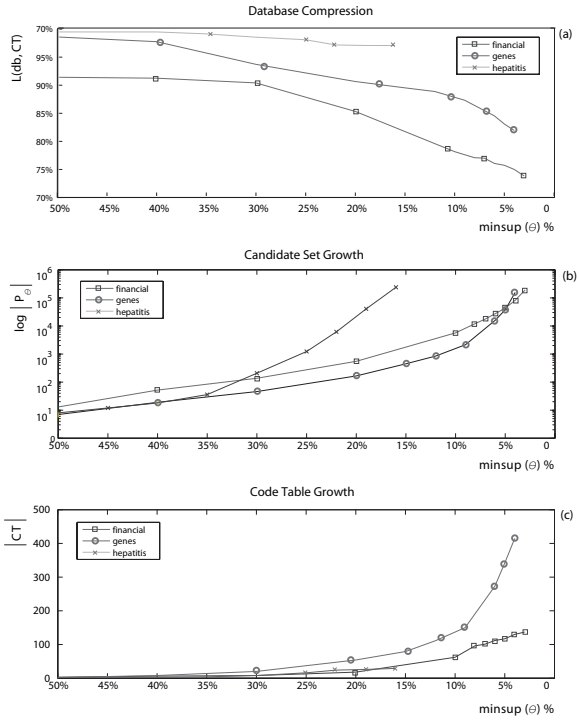

Figure 3: Results for different minimum support values θ: (a) The encoded length obtained for the database, (b) the number of frequent patterns, and (c) the number of code table patterns in CT.

5. EXPERIMENTS

To experimentally validate our approach we run experiments on publicly available relational data sets from previous KDD-cups (see Table 1). These databases are: the *financial* [1], *genes interaction* [2], and *hepatitis* [3] databases. We use a frequent attributed tree miner [2] to generate the frequent pattern sets, as our relational patterns can be represented as attributed trees.

5.1 Describing the Database

In order to find the optimal model of the database, RDB-KRIMP would ideally evaluate *all* patterns. However, in order to be efficient, we evaluate all *frequent* patterns. To measure the effect of the minimum support value, we generate a frequent candidate set P_θ for various θ. Given a P_θ we compress the database using RDB-KRIMP, which results in a code table CT and an encoded database size $\mathcal{L}(CT, db)$.

The effect of sweeping the minimum support is shown in Figure 3a. For all used databases, we see that increasingly lower encoded database sizes are obtained for lower minimum support values.

The smaller encoded database sizes relate to the larger available sets of candidate patterns (see Figure 3b). In all cases we see that this candidate set growth is exponential. These larger candidate sets contain more patterns that can be inserted in the code table to contribute to the database description.

While we see that P_θ grows exponentially for lower values of θ, we do not see this trend in the size of the code table (see Figures

[1] http://lisp.vse.cz/challenge/
[2] http://pages.cs.wisc.edu/ dpage/kddcup2001/
[3] http://lisp.vse.cz/challenge/

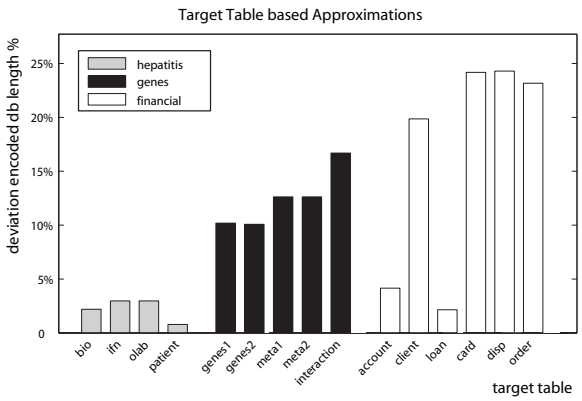

Figure 4: Choosing a particular table as a target decreases performance. For the lowest used minimum supports, the use of a target table leads to a worse encoding on all databases. Even for the star-shaped Hepatitis database, which seems well suited for a target table based approach.

3b and 3c respectively). Our code table grows for lower values of θ, but still only a small set is necessary to model the data. With respect to the original candidate set, our code tables achieve up to 4 orders of magnitude reduction in terms of number of patterns.

5.2 Initial Database Order

As described in Section 3, RDB-KRIMP re-orders the database in order to ensure a lossless encoding for the database. As the initial database order is not determined by our algorithm, it can potentially influence the resulting tuple order. Therefore, we need to evaluate the extend of its influence on the resulting database order and the resulting encoded database length. In order to evaluate the effect of this initial order, we randomly shuffle the tuples within the tables.

The experiments indicate that the influence on the resulting compression is minimal. The number of patterns used to encode the database is very similar to the original versions and lead to a similar database compression. The deviation for the database encoding for the *financial*, *genes*, and *hepatitis* database is respectively 1.12%, 0.35%, and 0.01%. Hence, RDB-KRIMP is robust with respect to the initial database order.

5.3 Fixing a Target Table

In our approach, we generalise the FARMER pattern language such that we do not rely on a specific target table. We expect a better description of the database by considering patterns starting from all possible tables. In order to evaluate this, we compare the results from alternative trials using a fixed target table as is usual in relational data mining [3, 13]. In these experiments, we encode the complete database using solely patterns that originate from a specific target table.

For all possible target tables, we determine how well we can approximate the original result that is obtained for the lowest used minimum support. We have depicted the deviation from the original result in Figure 4. We see that fixing the candidate set to a specific target table has a negative influence on the encoding of the database. The compression deteriorates up to 24% compared to the best obtained encoded size. This shows that allowing patterns to start at any table leads to a better description of the database.

The hepatitis database shows some additional interesting results. This database is a prime example of a target table based database,

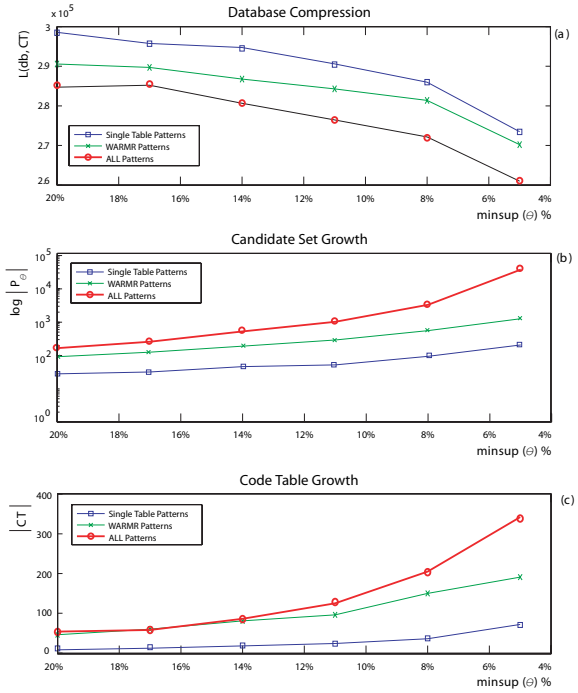

Figure 5: Our generalised relational patterns lead to better results. While the number of candidates grows large for low minimum supports (b), we obtain good database encodings (a) using compact code tables (c) (shown for the genes database).

as all (medical test) data surrounds a main table: $patient$. Even in this case, we see that picking a single target table leads to an increase in encoding length (a worse code table). Apparently, we can describe some data better when we do *not* pick the patient table as the single target table.

5.4 Comparison to Other Pattern Types

In Section 2, we defined a rich pattern language to match the database complexity. To measure whether or not we can describe the database better using more intricate patterns, we here compress the database with an increasingly more general pattern definition. A better database description would lead to smaller encoded database sizes.

A single complex pattern can describe structure in the database that would otherwise require multiple simpler patterns, possibly leading to a better compression. We have used the following characteristic pattern definitions to evaluate:

Single Table Patterns. Here we only allow $p = T^i(\{A_1, \ldots, A_j\})$ patterns in our P_θ. In other words, all candidate patterns only cover one table, and no joins are allowed. With this candidate set, we compress the database solely with single table patterns.

WARMR-like Patterns. In this candidate set, each pattern covers one tuple per table at most. In our notation, we define these patterns as: $p = p_0[p_1, \ldots, p_k]$. These patterns are similar to those used in WARMR-like approaches, which use an existential quantifier to select strictly one tuple from a table. Unlike WARMR applications, we do allow these patterns to start at any table in the database, in stead of a single target table [3].

Our language. Here we consider patterns as defined in Section 2. In this set, the patterns are defined as $p = p_0[[p_1^1, \ldots, p_{k_1}^1], \ldots, [p_1^q, \ldots p_{k_q}^q]]$. Recall that this is similar to grouping all FARMER

Table 2: On all databases more general patterns lead to smaller encoded sizes for the database.

	single table		WARMR		all	
	$\mathcal{L}\%$	$\#CT$	$\mathcal{L}\%$	$\#CT$	$\mathcal{L}\%$	$\#CT$
financial	91%	29	76%	130	76%	117
genes	87%	72	86%	191	83%	342
hepatitis	99%	5	98%	13	97%	26

generated pattern sets for each table.

Note that we order all three candidate sets as defined in Section 4. For all above scenarios, we evaluate the effect on the candidate set size $|P_\theta|$, the number of code table elements $|CT|$, and the compressed encoded size of the database $\mathcal{L}(CT, db)$.

The depicted result of the genes database in Figure 5a shows a typical result in terms of compressing the database. We see that the single table patterns lead to the worst encoded length for the database. A better compression can be derived when we allow WARMR patterns in our candidate set, which consequently can be improved by allowing all patterns to cover the database. We outline the obtained results on all databases in Table 2.

As before, we see that the number of candidate patterns grow very steeply for lower minimum support values. Also, we see that for more general pattern types we see a steeper candidate set growth (see fig. 5b). Note that we use a log scale to depict the number of patterns.

We have seen that achieving a smaller database encoding relies on the ability to draw patterns from an enriched pattern set. The inclusion of these enriched patterns indicates that essential characteristic structure within the database can be best described with these type of patterns: simpler patterns are apparently not sufficient. Although RDB-KRIMP steers to small code tables, it allows these additional patterns in the code table only if they aid in the description of essential structure that would otherwise be described in a much less efficient manner (see fig. 5c).

5.5 Relational Code Tables

Compactness is not the only feat of interest. In order for our model to provide insight in the database it should be interpretable. As our code table contains a collection of characteristic patterns, we can pick single code table patterns to examine.

To show an example, we pick a code table pattern that is intuitive without expert knowledge. Shown in Figure 6 is a pattern that reads as follows: *'A gene localised in the nucleus having a transcription function that has two distinct physical interactions'*. As one would expect, a characteristic pattern for the GENES database is that transcription often involves physical interactions in order to copy information and that it occurs in the nucleus. Note that this pattern selects two distinct tuples from the `INTERACTION` table.

The small encoded database lengths are obtained largely due to the availability of the more complex patterns in our pattern language. Patterns that select multiple tuples from one table are used more often in the database description, leaving simpler patterns to 'fill up' the database.

In the code tables obtained for the lowest used minsup, patterns that select multiple tuples from one table make up for 16%, 63%, and 71% of the content for the *financial*, *genes*, and *hepatitis* databases respectively.

```
P : GENE({ Localization = nucleus })
      [ [   INTERACTION({ Type = physical }),
            INTERACTION({ Type = physical }) ],
        [   META({ Function = transcription } ) ]
      ]
```

Figure 6: A partial description: a code table element from the description of the Genes database. Note that these patterns select multiple tuples from the same table.

6. RELATED WORK

Our code tables are models for a complete database. Similarly, Probabilistic Relational Models (PRM) are graph based models that can be applied to model relational databases [5, 9]. A PRM is one graph, in which attribute values are linked with their co-occurrence probability. This contrasts to our code table, which is not a single graph, but a collection of local tree-like patterns. Moreover, an attribute value can occur multiple times in different local models, if this aids in describing the database better. This means that we can regard an attribute value in different contexts, instead of one.

In the field of Relational Data Mining (RDM), ILP based approaches, such as the WARMR algorithm, allow for the discovery of relational patterns [3, 15]. As seen in our experiments, we obtain better database descriptions when we use our pattern language compared to WARMR-type patterns (see fig. 5).

The main application for a WARMR-like approach is when one is specifically interested in a target table, for example when trying to improve the classification scores on a target table given relational information [16]. This target table leads to an effective manner to prune the search space, and allows for aggregate functions to improve the efficiency [8]. An efficient implementation is FARMER [13], which in addition allows for a more general pattern language similar to the patterns used in this work.

In the work of both [4] and [12] the goal to find a different type of pattern: multi-valued dependencies (MVD) and functional dependencies (FD). These are 'global' patterns: a dependency is a 'higher order' pattern similar to a constraint on the relation, in contrast to code table patterns, which are local patterns.

6.1 R-KRIMP

In earlier work, we introduced R-KRIMP which finds patterns of the form: $[p_0, \ldots, p_n]$. These patterns are less expressive than the patterns used in RDB-KRIMP, and thus have less descriptive potential. R-KRIMP patterns are similar to the work of [6], who define these patterns as *simple conjunctive queries*.

In RDB-KRIMP a single code table element can describe more structure in the database than with R-KRIMP. As an example, consider a RDB-KRIMP-style pattern from our illustrative database: *'A account with frequency 3 having both a disponent-type disposition and a owner-type disposition'* (see fig. 2). This single patterns translates requires two individual R-KRIMP-style patterns: *'A account with frequency 3 having a disponent-type disposition'* and *'A account with frequency 3 having a owner-type disposition'*.

Note that R-KRIMP in this case would cover `ACCOUNT` $\bowtie$ `ORDER` in an overlapping manner. As tuples can occur multiple times within a join, R-KRIMP patterns allow for an overlapping cover. Thus, even in the initial case, when solely alphabet patterns are used, the covering of duplicate tuples will lead to a very different encoding.

7. DISCUSSION

In our experiments, we see that compared to the candidate set the code table growth shows a much slower pace.

Not only it stays small compared to the candidate set, it also stays compact compared to the original database. From the original set of frequent candidates only a few patterns contribute to the database description, leading up to 4 orders of magnitude reduction.

When we look at the influence of the initial database order, we see that this does not have much effect. We obtain only a slight deviation, up to around 1% from the original encoded length.

Enforcing a single target table rather than treating all tables equal yields a much worse database description. The encoded database length shows an increase of up to 25% compared to the original result.

In the *hepatitis* database all tables center around one single central table: the `PATIENT` table. While this database seems like a good case to pick as a target table, even here we obtain our best description when patterns are allowed to start at other tables.

Allowing a richer pattern language is a fruitful effort. We see that we generate more potentially interesting patterns in this manner, from which we can select those that describe the database best. Compared to WARMR-like patterns, we see that we can describe the database better: we achieve shorter encoded lengths. To obtain these good database descriptions, the code tables rely for the large part on these enriched patterns, which are of the type that selects multiple tuples from a table.

Our code tables stay compact and do not show exponential set growth as frequent pattern sets do for lower minimum supports. RDB-KRIMP shows to be successfully to select compact models. Even under its MDL-selection pressure, larger enriched patterns are selected to describe the database, indicating that these patterns are very characteristic for the database.

8. CONCLUSION

Currently, in order to obtain insight from a relational database, either one aims to find a *'global'* model of the complete database, or one seeks a collection of *'local'* patterns in the database. While both approaches have their merits, they have their shortcomings. Global models tend to blur out interesting local structure for the sake of the global structure, and pattern collections tend to drown the global picture in a sea of patterns. In this paper, we propose a method that combines the merits of both directions: it mines a compact but detailed description of the complete relational database.

For a database model to be descriptive it should be able to reflect the patterns that are present within it. Relational databases allow for tuples in one table to be connected to multiple tuples from other tables. Hence, if we want to find interesting patterns in a database, our pattern language should be rich enough to reflect such relations.

In this paper, we introduce RDB-KRIMP, an algorithm that describes the complete relational database using only a small collection of characteristic patterns: the code table. While the code table serves as a global model for the complete database, its patterns preserve the local details. Using the MDL principle, RDB-KRIMP results in a compact set of patterns that together describe the database well. With respect to the original set of frequent candidates, we obtain up to 4 orders of magnitude reduction.

Our rich relational pattern language allows RDB-KRIMP to draw from a larger pool of interesting frequent candidate patterns. The experiments reported on in this paper verify our claims both on the usefulness of our pattern language and on the ability of RDB-KRIMP to select just a few highly descriptive patterns. Firstly, each code table heavily relies on the introduced pattern language: up to 70% of the patterns select multiple tuples from one table. Secondly, experiments show that our pattern language leads to a far better compression. In other words, these patterns highlight characteristic structure in the database. The fact that these patterns are not only characteristic but also easily interpretable is shown by an example from the Genes database (see fig. 6). As part of the code table we find a pattern that describes gene interactions: *'A gene localised in the nucleus having a transcription function that has two distinct physical interactions'*.

In contrast to current frequent pattern mining approaches, our approach does *not* rely on a target table. That is, our patterns can start from any table. In all cases this yields to improvements, of up to almost 25%. Even in a typical 'target table' style database, we see that we obtain a better description without the use of a single table as target.

9. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.

[2] J. De Knijf. Fat-miner: mining frequent attribute trees. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 417–422, New York, NY, USA, 2007. ACM.

[3] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.

[4] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Commun.*, 12(3):139–160, 1999.

[5] L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[6] B. Goethals, W. Le Page, and H. Mannila. Mining association rules of simple conjunctive queries. In M. Zaki and K. Wang, editors, *SDM*, pages 96–107. SIAM, 2008.

[7] P. D. Grünwald. Minimum description length tutorial. In P. Grünwald and I. Myung, editors, *Advances in Minimum Description Length*. MIT Press, 2005.

[8] A. Knobbe. *Multi-Relational Data Mining*. PhD thesis, Universiteit Utrecht, Utrecht, the Netherlands, 2004.

[9] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 580–587, 1998.

[10] A. Koopman and A. Siebes. Discovering relational items sets efficiently. In M. Zaki and K. Wang, editors, *SDM*, pages 108–119. SIAM, 2008.

[11] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, 1993.

[12] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 12(1):83–99, 1994.

[13] S. Nijssen and J. N. Kok. Efficient frequent query discovery in farmer. In *In Proc. of the 7th PKDD, volume 2838 of LNCS*, pages 350–362. Springer, 2003.

[14] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In J. Ghosh, D. Lambert, D. B. Skillicorn, J. Srivastava, J. Ghosh, D. Lambert, D. B. Skillicorn, and J. Srivastava, editors, *SDM*, pages 393–404. SIAM, 2006.

[15] M. S. Tsechansky, N. Pliskin, G. Rabinowitz, and A. Porath. Mining relational patterns from multiple relational tables. *Decis. Support Syst.*, 27(1-2):177–195, 1999.

[16] X. Yin, J. Han, J. Yang, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, pages 399–411, Washington, DC, USA, 2004. IEEE Computer Society.