Mining Interesting Sets and Rules in Relational Databases

Bart Goethals
University of Antwerp
Department of Mathematics
and Computer Science
bart.goethals@ua.ac.be

Wim Le Page
University of Antwerp
Department of Mathematics
and Computer Science
wim.lepage@ua.ac.be

Michael Mampaey[*]
University of Antwerp
Department of Mathematics
and Computer Science
michael.mampaey@ua.ac.be

ABSTRACT

In this paper we propose a new and elegant approach toward the generalization of frequent itemset mining to the multi-relational case. We define relational itemsets that contain items from several relations, and a support measure that can easily be interpreted based on the key dependencies as defined in the relational scheme. We present an efficient depth-first algorithm, which mines relational itemsets directly from arbitrary relational databases. Several experiments show the practicality and usefulness of the proposed approach.

1. INTRODUCTION

Itemset mining algorithms are probably the most well-known algorithms in the field of frequent pattern mining. Many efficient solutions have been developed for this relatively simple class of patterns. While the task of mining frequent itemsets in a *single* relation is well-studied, only a few solutions exist for mining frequent itemsets in arbitrary relational databases, which typically have *more* than one relation [4,5,9,10]. These methods consider a *relational itemset* to be a set of items, where each item is an attribute-value pair, belonging to one or more relations in the database. In order for two or more items from different relations to be in the same itemset, they must be *connected*. Two items are considered to be connected if there exists a join of their two relations in the database that connects them. In general, an itemset is said to *occur* in the database, if there exists a tuple in a join of the relations, which contains the itemset. In this paper we also adopt this notion of occurrence.

A good definition of a unit in which the support of a pattern is expressed — i.e. what is being counted — is a primary requirement to mine any type of frequent pattern. In existing works on relational itemset mining [4,9,10], the frequency of an itemset over multiple relations is expressed in the number of occurrences in a join of the database's relations. However, this definition of itemset support is hard to interpret, because it heavily depends on how well the items in the set are connected. In this paper, we assume that key dependencies are specified in the relational scheme of the input database. We determine the support of an itemset by counting unique key values in the tuples where the itemset occurs. Consider the relational database in Figure 1, which we will use as a running example throughout the paper. For this database, the keys to be used are {Professor.PID, Course.CID, Student.SID, Study.YID}. This new support counting technique allows for interpretable frequent itemsets, as it goes without saying that itemsets frequent in Professor.PID have different semantics than itemsets frequent in Course.CID. This approach permits an efficient depth-first algorithm that generates interesting frequent relational itemsets that are easy to understand.

[†]An extended version of this paper is available at [7].

[*]Michael Mampaey is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

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.
SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

2. DEFINITIONS

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

2.1 Relational Scheme

Every relational database has a relational scheme. For the sake of clarity, we focus on simple relational schemes. More specifically, we consider acyclic schemes using only binary relations, i.e. schemes that can be represented as an unrooted tree. Let *sort* be a function that maps a relation name to its attributes [1]. We define such schemes as follows.

Definition 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. $\forall E_i, E_j \in \mathcal{E} : \exists! E_1, \ldots, E_n \in \mathcal{E}$ *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, and that for every two entities there is a unique path of binary relations and entities connecting them. Many realistic relational databases satisfy such simple relational schemes. Moreover, databases with star schemes and snowflake schemes can be formulated in this way.

2.2 Relational Itemsets

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

DEFINITION 2. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, the set $\{(A_1 = v_1), \dots, (A_n = v_n)\}_K$ 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 there is no $(A_j = v_j)$ with $j \neq i$ such that $A_j = A_i$. We denote the set of all items by $\mathcal{I}$.*

Given the relational database in Figure 1, let us abbreviate the relation names to P, C, Stnt and Stdy. The relational itemset $\{(\mathsf{P.Name} = \mathtt{Jan}), (\mathsf{C.Credits} = \mathtt{10})\}_{\mathsf{C.ID}}$ describes courses that have 10 credits and that are taught by professors named Jan, since C.ID is the key of the Courses relation, and professors can only be connected to courses in one way, via the relation Teaches.

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

PROPOSITION 1. *Given a simple relational scheme $(\mathcal{E}, \mathcal{R})$, and a relational itemset $I_K = \{(A_1 = v_1), \dots, (A_n = v_n)\}_K$, let $\mathcal{E}_{I_K} = \{E \in \mathcal{E} \mid \mathrm{key}(E) = K \text{ or } \exists i : A_i \in sort(E)\}$. There exists a* ***unique path*** *P_{I_K} connecting all entities $E \in \mathcal{E}_{I_K}$.*

DEFINITION 3. *Given an instance of a simple relational scheme $(\mathcal{E}, \mathcal{R})$, the* ***absolute support*** *of a relational itemset $I_K = \{(A_1 = v_1), \dots, (A_n = v_n)\}_K$ is the number of distinct values of the key K in the answer of the query (expressed here in relational algebra [1]):*

$$\pi_K \sigma_{A_1=v_1,\dots,A_n=v_n} E_1 \bowtie_{\mathrm{key}(E_1)} R_{1,2} \bowtie_{\mathrm{key}(E_2)} E_2 \bowtie \dots \bowtie E_m$$

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

PROPOSITION 2. *The support measure defined above is monotonic with respect to set inclusion, i.e. for all keys K and for all itemsets I_K, J_K it holds that*

$$I_K \subseteq J_K \Rightarrow supp(I_K) \geq supp(J_K).$$

The absolute support of the itemset $\{(\mathsf{C.Credits} = \mathtt{10})\}_{\mathsf{P.PID}}$ is 6, since the answer to the query $\pi_{\mathsf{P.PID}} \sigma_{\mathsf{C.Credits=10}} \mathsf{P} \bowtie_{\mathsf{PID}}$ Teaches $\bowtie_{\mathsf{CID}}$ C is $\{\mathtt{A,B,C,D,G,I}\}$. This means that six professors teach a course with 10 credits. The relative support is 6/9, as there are nine professors in the Professor relation.

2.3 Relational Association Rules

Association rules are defined just as in standard itemset mining. The only restriction is that the antecedent and the consequent need to be expressed in the same key.

DEFINITION 4. *Let $\mathcal{I}_K$ be the set of all relational items in key K. The rule $A \Rightarrow_K C$ is a* ***relational association rule*** *in key K if $A, A \cup C \subseteq \mathcal{I}_K$.*

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

Studies

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

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

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

Student

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

Takes

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

Study

YID	Name
I	Computer Science
II	Mathematics

Figure 1: Example of an instance of a simple relational scheme.

DEFINITION 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.*

Given Figure 1, an example of a relational association rule is $\{(\mathsf{P.Name} = \mathtt{Jan})\} \Rightarrow_{\mathsf{C.CID}} \{(\mathsf{C.Credits} = \mathtt{10})\}$. The confidence is $3/4 = 0.75$ since there are three courses $\{\mathtt{1,2,4}\}$ taught by a 'Jan' that have ten credits, compared to the four courses $\{\mathtt{1,2,3,4}\}$ taught by a 'Jan' in total. The relative support is $3/11 = 0.27$ since there are eleven courses.

3. ALGORITHM: SMURFIG

In this section we present two algorithms for mining relational itemsets. We first construct a naive algorithm based on the computation of the full outer join. Then we present the SMuRFIG algorithm (**S**imple **Mu**lti-**R**elational **F**requent **I**temset **G**enerator). Both algorithms employ KeyID lists, similar to the *tid* (transaction identifier) lists used in the well-known Eclat algorithm [12].

First, we consider the naive approach. The input of the Naive algorithm is an instance of a simple relational scheme and a relative support threshold *minsup*. The support query from Definition 3 is straightforwardly decomposed into three parts, i.e. a join, a selection, and a projection. First, a join table J is constructed, in which the correct supports are to be found. However, this join is different for each itemset, and performing all such possible joins is infeasible. Instead, we create a single large join table using all entities and relations. To construct J, we cannot use an equi-join. Indeed, if a tuple is not connected to any tuples in other tables, it does not appear in the full equi-join of all entity tables, which means we 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.

Then, a standard frequent set miner, Eclat, is applied to J using a new threshold. Rather than using the relative support threshold *minsup*, we must use a new absolute threshold $abssup = minsup \times \min_E |E|$ for J. The absolute support of an itemset in J is at least as high as the absolute support of that itemset for any key K, so any itemset frequent in some key K with respect to *minsup* will also be frequent in J with respect to *abssup*. In a way, the Eclat algorithm

fulfills the role of the select clause (σ) of the support query. We assume that the tid lists of the itemsets (the lists of tuples of J where the itemsets occur) are accessible to us. Finally, these tid lists are *translated* to their appropriate KeyID lists. Translating a tid list T to a KeyID list comes down to performing the projection $\pi_K(J \bowtie T)$ to each key K. This can be done efficiently by using lookup tables that can be created during the construction of J. At the end, the relative minimum support threshold is imposed for each itemset's KeyID list, and the frequent itemsets are reported.

The advantage of the Naive algorithm is that it can be implemented as a wrapper around an existing itemset mining algorithm. However, the computation of the full outer join can be expensive in both time and memory, making it infeasible to use on larger databases. Moreover, too many candidates are generated. We can only prune itemsets that are infrequent with respect to *abssup*, but many candidate sets that are frequent in J may turn out to be infrequent for all keys K with respect to the *minsup* threshold.

The SMuRFIG algorithm (see Algorithm 1) does not suffer from these disadvantages, i.e. it is efficient in both time and memory. It uses the concept of *KeyID list propagation*. First, the KeyID lists of all items are fetched from the data in their respective entities (Line 3), and then these KeyID lists are propagated to all other entities (Line 5). The propagation function recursively translates a KeyID list from one entity E_i to its adjacent entities E_j, until all entities are reached. Translating a KeyID list T_i from E_i to E_j via $R_{i,j}$ is equivalent to executing the relational query $\pi_{key(E_j)}(T_i \bowtie R_{i,j})$. It is easy to verify that we now have the KeyID lists of all items for all keys K, and hence their supports.

Next, the (frequent) singleton itemsets are combined into larger sets by the *Keyclat* function (Algorithm 2), which is the core of SMuRFIG. The search space is traversed depth-first. In each recursion, two k-itemsets I' and I'' with a common prefix P (initially empty) are combined to form a new candidate set $I = I' \cup I''$ of size $k+1$. To compute the support of I, we first determine that the entity tables of the suffixes of I' and I'' are E_1 and E_2 (Line 4). We 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 (Line 7), and then these KeyID lists are propagated to all other entities (Line 8). For these remaining entities E with key K, it does not necessarily hold that $\text{KeyIDs}(I'_K) \cap \text{KeyIDs}(I''_K)$ results in $\text{KeyIDs}(I_K)$. We must additionally intersect this with the propagated KeyID lists, in order to obtain the final KeyID list of I_K in E (Line 11). Some further optimizations, as well as a proof of correctness of SMuRFIG are provided in [7].

Algorithm 1 SMuRFIG: 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 frequent itemsets I

```
1: for all E ∈ 𝓔 do
2:   K := key(E)
3:   𝓘_E := Singletons(E)
4:   for all I_K ∈ 𝓘_E do
5:     Propagate(KeyIDs(I_K))
6:     if ∃K' : supp(I_K') ≥ minsup × |E| then
7:       𝓘 := 𝓘 ∪ {I}
8: 𝓕 := Keyclat(𝓘, minsup)
9: return 𝓕
```

Algorithm 2 Keyclat: computation of itemsets' KeyID lists

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

```
1: for I' in 𝓛_P do
2:   for I'' in 𝓛_P with I'' > I' do
3:     I := I' ∪ I''
4:     E_1, E_2 := entity of suffix items I' \ P, I'' \ P resp.
5:     for i ∈ {1,2} do
6:       K_i := key(E_i)
7:       KeyIDs(I_{K_i}) := KeyIDs(I'_{K_i}) ∩ KeyIDs(I''_{K_i})
8:       pKeyIDs_i(I) := Propagate(KeyIDs(I_{K_i}))
9:     for E ∈ 𝓔 \ {E_1, E_2} do
10:      K := key(E)
11:      KeyIDs(I_K) := pKeyIDs_1(I_K) ∩ pKeyIDs_2(I_K)
                         ∩ KeyIDs(I'_K) ∩ KeyIDs(I''_K)
12:      supp(I_K) := |KeyIDs(I_K)|
13:      if supp(I_K) ≥ minsup × |E| then
14:        𝓕_I' := 𝓕_I' ∪ I_K
15:  𝓕 := 𝓕 ∪ 𝓕_I' ∪ Keyclat(𝓕_I', minsup)
16: return 𝓕
```

The time complexity of SMuRFIG is as follows. Per 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 is executed at most twice and uses each relation R once, amounting to $O(\sum_{R\in\mathcal{R}}|R|)$. Hence, the time complexity of SMuRFIG is $O\left(|\mathcal{F}| \cdot size(\mathcal{DB})\right)$, 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. When an itemset of length k is generated, we have $(k^2+k)/2$ previous itemsets in memory due to the depth-first traversal. For each of these itemsets, KeyID lists are stored 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 $O\left(l^2 \cdot \sum_{E\in\mathcal{E}}|E| + \sum_{R\in\mathcal{R}}|R|\right)$.

4. SUPPORT DEVIATION

The relational setting that we are considering also brings additional challenges. For instance, let us consider the itemset $\{(\mathsf{C.project} = \mathtt{Y})\}_{\mathsf{S.SID}}$ having a support of 67%. For a lower support threshold this would be a frequent itemset, but it is not necessarily an *interesting* one. Suppose that we also find that $\{(\mathsf{C.project} = \mathtt{Y})\}$ 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 one course then the expected support (if one assumes no bias) would be 30%. However, if they take two courses each, it rises to 51.2%, for three courses this becomes 66.1%, etc. So in the case of an average of three courses per student, a support of 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 C.CID to compute the *expected support* of $\{(\mathsf{C.project} = \mathtt{Y})\}_{\mathsf{S.SID}}$, and we discard the itemset if its real support does not deviate substantially from its expected support.

To formalize this notion, we start off by only considering *intra-entity* itemsets, i.e. relational itemsets I_K consisting of items from a single entity E with key K. We 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 of expected value) as follows.

DEFINITION 6. *Let $I_K \subseteq \mathcal{I}$ be an intra-entity itemset containing items from a single entity E with key K, and let K' be the key of some other entity E'. Then the **expected absolute support** of $I_{K'}$, given that $supp(I_K) = s$, equals*

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

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.*

The derivation of this formula is omitted due to the lack of space, but can be found in [7]. Using this definition of expected support, we formally introduce *deviation*.

DEFINITION 7. *Given an itemset $I_K \subseteq \mathcal{I}$, let E_K be the entity of key K. We define the **deviation** of I_K as*

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

Our goal is to find frequent relational itemsets with a given minimal deviation, in order to eliminate redundant relational itemsets. An experimental evaluation can be found in the next section. Note that we restricted ourselves to a special case of intra-entity itemsets. In order to generalize deviation and expected support to itemsets containing attributes from multiple entities, which we refer to as *inter-entity* itemsets, it is necessary to generalize the definition of itemset support to allow sets of keys. This would also entail substantial changes to our algorithm and is therefore part of our future work.

5. EXPERIMENTS

In this section we consider a summary of the results of several experiments we performed on real world databases. A more extensive experimental evaluation can be found in [7]. The SMuRFIG algorithm[1] was implemented in C++, and run on a system with a 2.16 GHz processor and 2GB RAM.

First, we consider a snapshot of the student database from the University of Antwerp's Computer Science department. The scheme roughly corresponds to the one of Figure 1. 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 KDD-Cup 2003[2], and comprises a large collection of High Energy Physics (HEP) papers. It consists of HEP papers linked to authors, journals, and also to other papers (citations). It contains 2543 papers, 23621 authors and 76 journals, and there are 5012 connections between authors and papers, plus 458 links from papers to journals.

[1] http://www.adrem.ua.ac.be/

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

5.1 Patterns

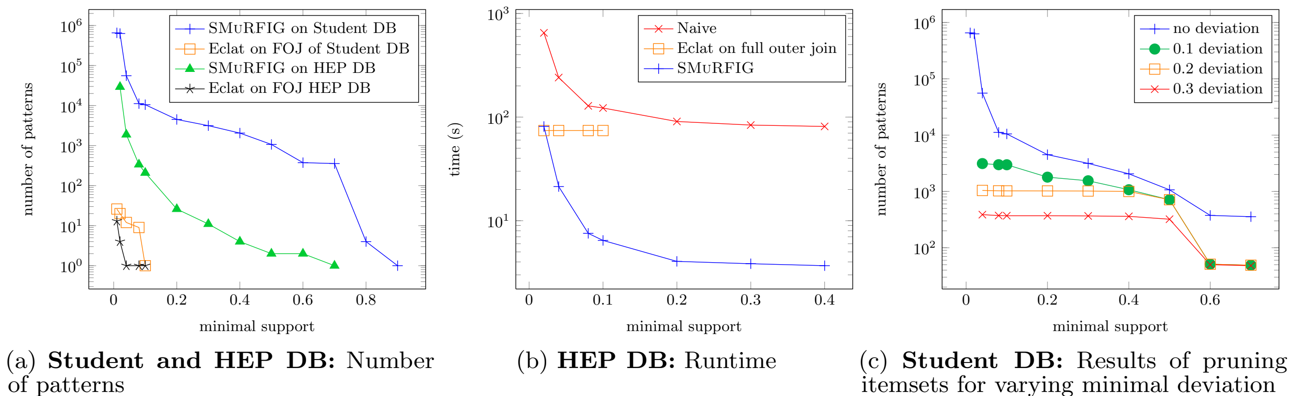
Several interesting patterns were discovered in the Student database, and we now discuss a few of them. We found $\{(\mathsf{S.Study} = \texttt{1-MINF-DB})\}_{\mathsf{P.PID}}$ with a support of 63%, stating that 63% of the professors teach a course that is taken by students of `1-MINF-DB`. We also find 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`. This is a clear example of the merit of key-based frequency. Some patterns found in the HEP database include the following association rules: $\{(earliest_year{=}2002)\} \Rightarrow_{\text{paper.id}} \{(published{=}\texttt{false})\}$ and $\{(earliest_year{=}2003)\} \Rightarrow_{\text{paper.id}} \{(published{=}\texttt{false})\}$ with respectively 60% and 93% confidence. Since the dataset is from the 2003 KDD-Cup, this tells us that most recently submitted papers ($earliest_year > 2002$) are not yet published. These examples demonstrate that relational itemsets and rules with key-based frequency allow us to find interesting, easily interpretable patterns.

5.2 Performance

We experimentally evaluated the performance of the Naive and SMuRFIG algorithms on the Student and HEP datasets, gradually varying the minimum support threshold. In our experiments we also ran a standard categorical Eclat algorithm [3] on the full outer join table, an approach taken in previous works. The number of patterns and the runtimes are reported below. In Figure 2a 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 Figure 2b 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 latter operates on an extremely large data table, while SMuRFIG works with smaller tables from the original database. Note that SMuRFIG is also faster than Eclat on the join table, even though the latter finds far fewer patterns. To evaluate the deviation measure we carried out experiments on the Student DB with varying support and deviation thresholds. As is apparent in Figure 2c, increasing the deviation threshold effectively reduces the number of itemsets. Scalability experiments confirming our complexity analysis are reported in [7].

6. RELATED WORK

Our work is related to more general frequent query mining algorithms such as Conqueror [8] and WARMR [6]. Since the pattern types considered there are more general and complex, and especially since they use different support measures, a direct and fair comparison cannot easily be made. On the other hand, we are not aware of many approaches considering the more specific case of (categorical) itemsets in a multi-relational setting. Koopman and Siebes' R-KRIMP [9], and Crestana-Jensen and Soparkar's algorithm [4] both use the number of occurrences of the itemset in the join of all tables as the support measure. This is done without fully computing or



(a) **Student and HEP DB:** Number of patterns

(b) **HEP DB:** Runtime

(c) **Student DB:** Results of pruning itemsets for varying minimal deviation

Figure 2: Experimental Results for the Student and HEP Databases.

at least storing this join. Some existing techniques are used; Crestana-Jensen and Soparkar is based on Apriori [2] and R-KRIMP is based on KRIMP [11], both with the addition of some case-specific optimizations. These algorithms return the same results as Apriori (∼KRIMP) run on the full join. This is not the case for our algorithm, since we compute supports simultaneously for multiple keys. Furthermore, compared to the two-phased approach of Crestana-Jensen and Soparkar, the computation of intra- and inter-entity itemsets is merged in our algorithm, and is performed in a depth-first manner. Ng et al. [10] focus on star schemes, in which case the join table is essentially already materialized as the fact table. This approach closely mirrors the two-phased method of Crestana-Jensen and Soparkar, but adds some optimizations specific to star schemes. None of these relational mining techniques take into consideration the semantic consequences of the blow-up that occurs in the joining of all tables. Cristofor and Simovici [5] do define a support measure similar to ours called *entity support*. For those itemsets consisting of attributes from a single 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. Furthermore, their proposed Apriori-based algorithm is only defined for star schemes, and explicitly computes joins, which does not scale well to larger databases.

7. CONCLUSIONS AND FUTURE WORK

In this paper we defined frequent relational itemsets using a novel support measure, based on the keys of the relational database scheme. We implemented an efficient depth-first propagation-based algorithm for mining these frequent relational itemsets. Our experiments showed that the SMuRFIG algorithm performs well on real datasets, is scalable, and is capable of finding interesting patterns. This stands in contrast with most existing work, where the support measure being used is unintuitive or inefficient to compute. In addition, we defined the deviation measure in order to address the statistical pattern blow-up that is specific to the relational context. Part of our future work is to study this measure further and extend it to inter-entity itemsets.

For the sake of clarity, we restricted ourselves to simple schemes. This already allows us to mine a large collection of databases (or parts thereof), and can result in interesting discoveries, as we have seen in Section 5. Preliminary experiments show that some small extensions to our definitions allow us to find more complex patterns while only suffering a small loss of efficiency. The study of the algorithmic implications of such extensions is part of our future work.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD 1993*, pages 207–216, 1993.

[3] T. Calders, B. Goethals, and M. Mampaey. Mining itemsets in the presence of missing values. In *Proceedings of ACM SAC 2007*, pages 404–408, 2007.

[4] V. Crestana-Jensen and N. Soparkar. Frequent itemset counting across multiple tables. In *Proceedings of PAKDD 2000*, pages 49–61, 2000.

[5] L. Cristofor and D. Simovici. Mining association rules in entity-relationship modeled databases. Technical Report, University of Massachusetts Boston, 2001.

[6] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1): pages 7–36, 1999.

[7] B. Goethals, W. Le Page, and M. Mampaey. Mining interesting sets and rules in relational databases. Technical Report 09.02, University of Antwerp, 2009.

[8] B. Goethals, W. Le Page, and H. Mannila. Mining association rules of simple conjunctive queries. In *Proceedings of SDM 2008*, pages 96–107, 2008.

[9] A. Koopman and A. Siebes. Discovering relational item sets efficiently. In *Proceedings of SDM 2008*, pages 108–119, 2008.

[10] E. Ng, A. Fu, and K. Wang. Mining association rules from stars. In *Proceedings of ICDM 2002*, pages 322–329, 2002.

[11] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *Proceedings of SDM 2006*, pages 393–404, 2006.

[12] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of KDD 1997*, pages 283–286, 1997.