Optimizing subset queries: a step towards SQL-based inductive databases for itemsets [*]

Cyrille Masson
INSA de Lyon-LIRIS
F-69621 Villeurbanne, France
cyrille.masson@liris.cnrs.fr

Céline Robardet
INSA de Lyon-PRISMA
F-69621 Villeurbanne, France
crobarde@lisi.insa-lyon.fr

Jean-François Boulicaut
INSA de Lyon-LIRIS
F-69621 Villeurbanne, France
jean-francois.boulicaut@liris.cnrs.fr

ABSTRACT

Storing sets and querying them (e.g., subset queries that provide all supersets of a given set) is known to be difficult within relational databases. We consider that being able to query efficiently both transactional data and materialized collections of sets by means of standard query language is an important step towards practical inductive databases. Indeed, data mining query languages like `MINE RULE` extract collections of association rules whose components are sets into relational tables. Post-processing phases often use extensively subset queries and cannot be efficiently processed by SQL servers. In this paper, we propose a new way to handle sets from relational databases. It is based on a data structure that partially encodes the inclusion relationship between sets. It is an extension of the hash group bitmap key proposed by Morzy et al. [8]. Our experiments show an interesting improvement for these useful subset queries.
Keywords: inductive databases, itemset post-processing.

1. INTRODUCTION

One of the most popular data mining task concerns the constraint-based discovery of itemsets and association rules from transactional data [1, 3, 9, 14]. In the so-called *inductive database approach* [6], a database integrates raw data with knowledge extracted from raw data, materialized under the form of patterns into a unified framework [4]. A knowledge discovery process is then considered as a querying process. Among the various proposals of a query language for association rule mining [2], the `MINE RULE` proposal [7] is typical: it allows to query databases using SQL but also to query association rules from the selected data. The evaluation of a `MINE RULE` query leads to the computation of a collection of rules whose components, i.e., itemsets, are stored in relational tables. However, most of the available query languages poorly support the needed post-processing steps for real-life association rule mining processes [2] and users often have to use standard query languages to search relevant patterns. In gene expression data analysis, for instance, once the frequent sets of co-regulated genes are available, it can take months to biologists before finding relevant patterns among them. Indeed, huge amounts of patterns can be extracted, even from small data sets [12].

*Research partially funded by the cInQ project IST 2000-26469

In this paper, we assume that the a priori interesting itemsets have been computed and stored in a relational database. Transactions (data) is also stored in the database and is also a set of itemsets. In practice, the patterns are associated to some properties like frequency or closeness. Most of the post-processing techniques need to access both data and patterns, and often involve a lot of set manipulations, typically *subset querying*. Crossing-over queries returning transactions satisfying some association rules are of this kind: transactions are itemsets that have to be supersets of some extracted itemsets. Another application where inclusion tests are used is the regeneration of the frequent itemsets from their so-called *condensed representations* [3]. Other examples of post-processing are given in [13].

In the field of relational databases, set-related operations are difficult tasks. We focus here on subset queries, i.e. in finding in which sets a given itemset is included. Subset queries can be expressed by means of the relational division operator, but SQL does not implement it, thus relying on queries involving multiple joins. Results of Section 4 confirm the fact that these queries are often not well suited for subset querying. Notice that we do not address here the use of SQL for mining the frequent itemsets [11, 10] that also needs for subset query evaluation. Figure 1 shows an example of SQL query retrieving the supersets of the itemset $\{5, 8\}$.

Itemset table

set_id	item_id
1	2
1	5
1	8
2	10
2	8
2	7
3	4

```
SELECT a.itemset_id
FROM Itemset a, Itemset b
WHERE a.set_id = b.set_id
AND a.item_id = 5
AND b.item_id = 8
```

SQL result

itemset_id
1

Figure 1: Retrieving supersets in SQL

Morzy [8] has proposed an efficient method for storing and manipulating itemset patterns in relational databases. It relies on the use of bitmap keys that summarize in a string the content of an itemset. We propose here a new method to store and index itemsets and bitmap keys so as to speed up subset queries. It is based on a data structure that partially encodes the inclusion relationship between itemsets. Our experiments on synthetic and real data show an interesting improvement for subset queries. We present the hash group bitmap keys method of [8] in Section 2, and our technique in Section 3. Experiments are described in Section 4.

2. HASH GROUP BITMAP KEYS

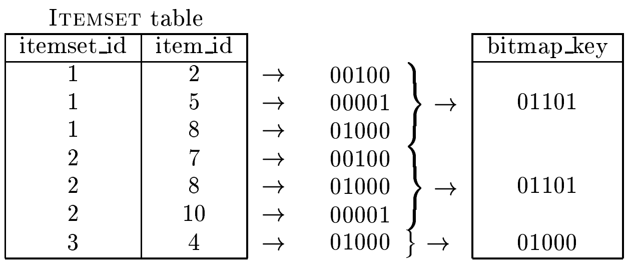
To optimize subset queries, [8] has introduced the idea of a bitmap key associated to each itemset of the database and which summarizes the content of the itemset. We first recall the notion of group bitmap key and then the principles of the hash group bitmap key. In this section, we suppose that we have N possible items in the different itemsets.

The group bitmap index for a given itemset S is a binary number of length N in which a bit at position k has the value '1' if and only if $k \in S$. All the different group bitmap keys are then stored in an index table together with an identifier of the set they encode. For instance, the bitmap keys associated to itemsets $\{0, 3, 5, 9\}$, $\{2, 5\}$, $\{1, 4, 6\}$ are respectively 1000101001, 100100, 1010010. When a subset query is performed, a bitmap key B is computed for the searched itemset, then the subset containment is checked by means of a bitwise AND operation between B and the set of keys in the database. So, for each bitmap key in the database, we just have to check if there is a 1 at each position where there is a 1 in B. For instance, if we search for supersets of $\{4, 6\}$ in the itemsets of the previous example, we will first compute the bitmap key associated to $\{4, 6\}$, i.e. 1010000, and then compare it by an AND operation with each itemset. We thus find that $\{1, 4, 6\}$ is a superset of $\{4, 6\}$.

This definition of bitmap keys has mainly a theoretical interest. Indeed, as soon as N becomes quite large (and this is the case when considering data mining processes), storing all the bitmap keys becomes space-consuming, because each key is N-bit long. Moreover, this technique is not suited for a dynamic context, when new items can be added through different experiments. Indeed, let us suppose that a new item is inserted, then we have to update the length of all the binary keys in the same way. Thus, the maintenance of these kind of index will be costly and difficult. That is why, in [8] Morzy and Zakrzewicz have proposed the hash group bitmap index. The idea is to consider binary keys of a fixed length n where $n << N$. The hash group bitmap key of an itemset is created with all the hash keys of the items that it contains. The hash key of an item X is an n-bit binary string computed as follows: $hash\_key(X) = 2^{X \ mod \ n}$ Figure 2 illustrates this hash group bitmap key computation for the itemset set $\{\{2, 5, 8\}, \{7, 8, 10\}, \{3, 4, 8\}\}$ with $n = 5$.

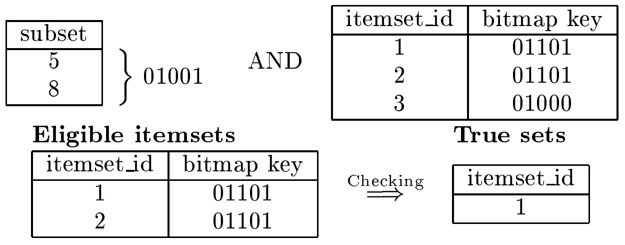
For a searched itemset X, a subset query is then evaluated in two steps: (1) First, we compute the hash group bitmap key of X and compare it with the hash group bitmap index of each itemset in the database, by means of a bitwise AND operation, and we return the identifiers of the itemsets satisfying this test. (2) Then, we have to check if the identifiers returned in the first step really correspond to supersets of X. Indeed, as an hash key can encode different itemsets (because of the hashing technique), it is possible to have some false positive. Therefore, we must verify this point, and this is done by using a traditional inclusion test. Figure 3 shows the evaluation of a subset query for the itemset $\{5, 8\}$.

The idea of a signature of an itemset under the form of



itemset_id	item_id			bitmap_key
1	2	→	00100	
1	5	→	00001	01101
1	8	→	01000	
2	7	→	00100	
2	8	→	01000	01101
2	10	→	00001	
3	4	→	01000	01000

Figure 2: Example of computation for $n = 5$.



subset
5
8

itemset_id	bitmap key
1	01101
2	01101
3	01000

itemset_id	bitmap key
1	01101
2	01101

itemset_id
1

Figure 3: Search with Hash Group Bitmap Keys

a bitmap key is useful, but one of main drawbacks of this method is that the search of supersets is done in a "blind" way. Indeed, each time we are looking for supersets of a given itemset, we have to check all the bitmap keys to find them. It lacks of an indexing method on itemsets and bitmap keys that can partially encode an inclusion relationship between itemsets, thus allowing to speed up the scan of the keys. That is what we are proposing in the next section.

3. A NEW STORAGE STRUCTURE

We propose a new method to store itemsets in relational databases. It encodes the inclusion relationship between itemsets by storing an itemset tree [5] in a relational table. To improve the comparison between itemsets, associated bitmap keys are also stored in the relational table.

Itemset trees have been proposed by A. Hafez et al. to incrementally store itemsets when computing frequent and valid association rules [5]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be an ordered set of items. Each node s of the tree T represents either an encountered itemset, or a subset of it, both ordered according the definite order on I. Let us consider two itemsets $s_i = \{a_1, \ldots, a_k\}$ and $s_j = \{b_1, \ldots, b_l\}$. We denote $s_i \leq_o s_j$, iff $\forall p \in \{1, \ldots, \min\{k, l\}\}$, $a_p \leq b_p$. s_i is an ordered subset of s_j, denoted by $s_i \subset_o s_j$, iff $s_i = \{a_1, \ldots a_k\}$, $s_j = \{a_1, \ldots, a_k, b_{k+1}, \cdots, b_l\}$ and $k < l$. Inserting an itemset s in T proceeds incrementally and recursively. The root node r represents the empty set $\{\}$. An itemset is inserted by examining the children of the root node r, ordered by the $\leq_o$ relation. The recursive insertion procedure considers the five following cases:
1 - If none of the children s_j of r have the same leading element than s, then s is inserted as a son of r and the insertion procedure ends.
2 - If there is a son s_j of r s.t. $s_j = s$, then the procedure ends.
3 - If there is a son s_j of r s.t. $s \subset_o s_j$, then s is inserted as a son of r and as a parent of s_j. Then the procedure ends.

4 - If there is a son s_j of r s.t. $s_j \subset_o s$, then the insertion procedure is recursively called with s_j as the tree root.
5 - If there is a son s_j of r s.t. s and s_j are sharing some of their leading elements ($s \cap_o s_j \neq \emptyset$[1]), then two nodes are inserted: a node $s_i = s \cap_o s_j$ as a son of r and a parent of s_j and a node s as a son of s_i. Then the procedure ends.

Figure 4 shows an example of the construction of an itemset tree for set of itemsets $\{\{1, 2\}, \{4, 6\}, \{1, 3, 5\}\}$.

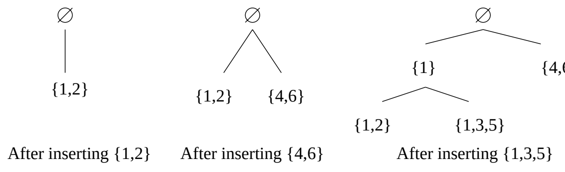


Figure 4: Exemple of itemset tree

We want to store the itemsets such that the ordered subset relationships that exist among them are preserved. We also store in the itemset tree the hash group bitmap key associated to each itemset because of the efficiency of the key scanning during subset querying. It leads to the following relational schema: (*IS_Id int*, *Nb_Items int*, *HBitmap_Key string*, *Ancestor int*, *First_Child int*, *First_Sibling int*, *Pattern bool*). Each row stands for an itemset or a subset of an itemset. In the first case, the PATTERN field takes the value "yes", "no" otherwise. Each row is identified by a unique number ITEMSET_ID which is used as an identifier in the field ANCESTOR which points to the parent node, in the field FIRST_CHILD which points to the first son of the current node, and in the field FIRST_SIBLING which points to the first neighbor of the current node. The field NB_ITEMS contains the number of items of the itemset. The hash group bitmap key of the itemset is stored in the field HBITMAP_KEY and the itemset is encoded in a separate table. We also store complete itemsets in a classical table of schema (*IS_Idint*, *itemint*). Figure 5 gives the encoding of the itemset tree for the set of itemsets $\{\{1, 2\}, \{4, 6\}, \{1, 3, 5\}\}$.

IS_Id	Nb_Items	HBitmap_Key	Ancestor	First_Child	First_Sibling	Pattern
1	0	00000	NULL	4	NULL	No
2	2	00110	4	NULL	5	Yes
3	2	10010	1	NULL	NULL	Yes
4	1	00010	1	2	3	No
5	3	01011	4	NULL	NULL	No

Figure 5: Encoding an itemset tree in a table

We use the itemset tree structure to solve subset queries without scanning all the rows of the itemset table. Algorithm 1 presents the principle of the resolution of the subset query: *node* is a node of the itemset tree and corresponds to a record of the table encoding itemset tree, the field *itemset* denotes the associated itemset. The function is recursive: for one node, it scans all its siblings. When a sibling is found as being a superset of a part of the subset, a recursive call is done on the first child of this sibling. The use of bitmap keys to check if a set is a superset of the subset can be used there. The depth search is stopped when we find in the subset an item not occuring in the candidate superset, whereas a bigger item is found. The fact that items are ordered is useful there to simplify this checking. When we find a node in which the subset is included, we automatically add to the result set all the children of the node that are marked as itemset. It is done by the function $Itemsets\_in\_Subtree$.

Algorithm 1: Search_Tree

```
Data : Subset is an itemset, node is a node of the tree
// Subset is a subset of the searched supersets
current = node; // e.g. node={1,3,5}
repeat
    brother = current.first_sibling;
    if Subset ⊆ current.itemset and current.is_itemset=yes then
        // we can add all the itemsets of the subtree
        results = results ∪ Itemsets_in_Subtree(current);
    else
        if ∀a ∈ Subset s.t. a ∉ current.itemset, ∄ b ∈ current.itemset, a < b then
            //Recursive call;
            results=results∪Search_Tree(current.first_child);
        end
    end
    current = brother;
until current ≠ null;
return results
```

4. EXPERIMENTS

We performed experiments on both synthetic and real data sets. Our experiments have been done on top of the MySQL DBMS. We used three different methods to search for supersets of given itemsets in the database: (1) the itemset tree method (cf. Section 3), (2) the Hash Group Bitmap Key index method [8], (3) a classical SQL query. We used the IBM Quest generator[2] to produce synthetic data with the following parameters: *n_trans* (the number of itemsets) was set to 20000, *n_items* (the number of different items) to 1000, *n_pats* (the number of patterns) to 10000, *patlen* (the average length of maximal patterns) to 4 and *corr* (the correlation between patterns) to 0.25.

First, we have looked at the behavior of the three methods w.r.t. the size of the searched set. We used a synthetic dataset with t_{len} (the average number of items per itemset) set to 15. Then we have searched 5 itemsets of differents sizes with the three methods (hash group bitmap key size was 29). Notice that searched itemsets were present in the database, so that the hash group bitmap key method is not disadvantaged, as it always need to scan all the stored keys, whereas the two others methods would quickly stop with sets not occuring in the data. Figure 6 shows the results.

A first remark is that SQL is quite fast when searching small itemsets, because of SQL server optimizations that "push" constraints on the presence of some items, thus pruning many irrelevant joins. The hash group bitmap key technique is efficient for small itemsets, because in these cases,

[1] $s_i \cap_o s_j = \{a_1, \cdots, a_k\}$ iff $s_i = \{a_1, \cdots, a_k, a_{k+1}, \cdots, a_p\}$ and $s_j = \{a_1, \cdots, a_k, b_{k+1}, \cdots, b_q\}$ with $a_{k+1} \neq b_{k+1}$

[2] `http://www.almaden.ibm.com/cs/quest/index.html`

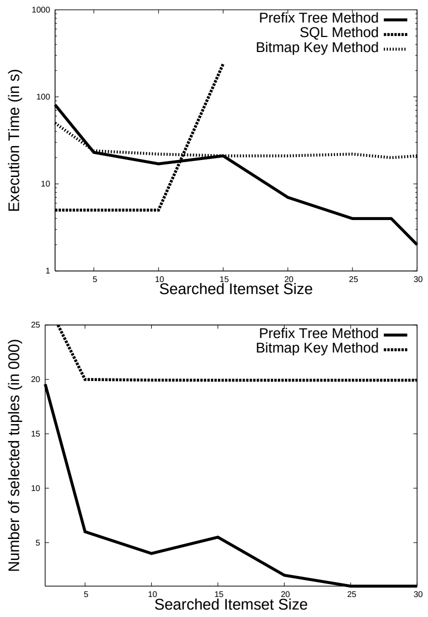


Figure 6: Experiments with synthetic datasets

the bitmap keys are discriminant enough to filter out many itemsets. However, with bigger searched sets, different items are likely to match the same bits of the key, thus leading to more false positive. Our itemset tree-based method appears as being efficient. The execution times are generally better than the ones of the two other methods and its execution time continues to drop with the increasing size of the searched set. On the contrary, the execution time of the hash group bitmap key method remains stable after some threshold, because this method involves a lower bound on the needed number of scans. Thus, if we store N itemsets, we will always scan at least N bitmap keys. With the itemset tree-based method, if the itemset tree is encoded using M tuples, we will scan at most M tuples ($M \geq N$) and, in the best case, just one tuple (the tree root). The execution time also drop because the larger the searched set is, the less numerous are the itemsets in which it can be included. Figure 6 also shows the number of selected tuples for the different methods and it confirms the theoretical intuition: the amount of tuples analyzed by our method drops with the increasing size of the searched set, whereas the Bitmap Key method reaches a threshold under which it cannot drop.

In a second experiment, we have analyzed how the execution time evolves w.r.t. the average size of stored itemsets. We generated different datasets with the same parameters than in the first experiment, except for t_{len} which was varying between 5 and 30. The size of the hash group bitmap key was 97. We searched a set of 90 itemsets of size 12. The results are given in Figure 7 (the curve of the SQL method is not depicted because of high execution times). It shows that our method is scalable with the increasing size of the sets in which the search is done, an important point for real iterative KDD processes, where it is difficult to know *a priori* the size of the itemsets. The cost of the Hash Group Bitmap Key method becomes prohibitive with big itemset sizes, because the size of the bitmap key is fixed whereas the number of items to encode grows. Thus, in these cases, two different items are more likely to match the same bit of the key, leading to more useless checks on the real sets.

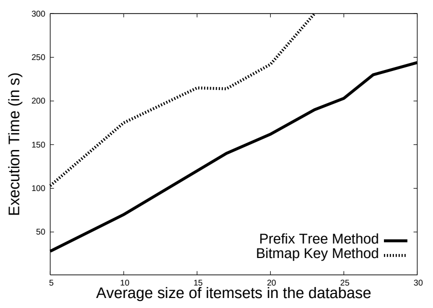


Figure 7: Execution time vs. average itemset size

Then, we have performed an experiment on the Census dataset of the UCI web site[3]. To show the relevancy of our approach in a KDD process, we have extracted all frequent closed sets (with a threshold of 30 %) and we obtained about 17000 itemsets of a maximal size of 13. We chosed a bitmap key size of 57 (one tenth of the number of possible items). We stored them in a database using the two methods described in Sections 2 and 3. We searched 10 itemsets of variable sizes, all of them were present in the dataset. The results are depicted on Figure 8. We can notice that the performance of the hash group bitmap key method was worst than the SQL method. Like already noticed on synthetic datasets, the basic SQL method is efficient with small searched subsets. However, when the size of the searched itemset grows, the itemset tree method is more efficient.

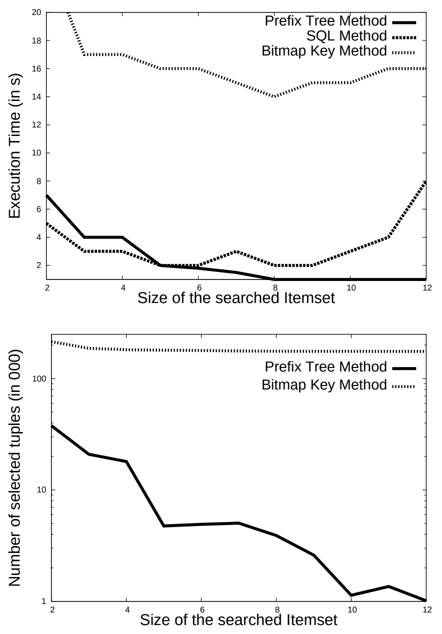


Figure 8: Results with the Census dataset

Finally, we have realized the same experiments on dis-

[3]http://www.ics.uci.edu/ mlearn/MLSummary.html

cretized microarray data. We have stored in the relational database all closed set in this data (10098 itemsets). Again, we applied the three methods on 5 itemsets of different sizes. There was 162 possible items and the average size of the itemsets was 5.25. The bitmap key size was set to 29 to keep the same proportion than in [8]. We have looked at the execution times and the numbers of selected tuples (cf. Figure 9). It confirms the previously obtained results. The Hash Group Bitmap Key method reaches a threshold under which it cannot go, whereas the itemset tree-based method is getting faster as the size of the searched sets grows.

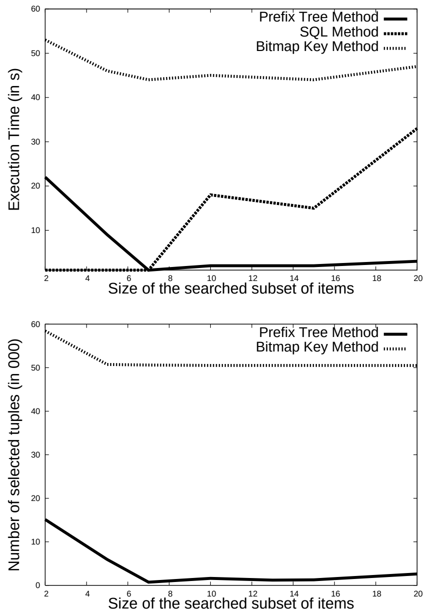


Figure 9: Experiments on real gene expression data

5. CONCLUSION AND PERSPECTIVES

We have considered the concept of inductive databases where raw data and patterns are stored in a common database framework. We know that set-related operations, and more particularly, subset querying is a critical aspect of pattern manipulation. That is why we have proposed a new method for storing and indexing itemsets to speed up subset queries. Our experiments on synthetic and real datasets have shown the relevancy of our approach w.r.t. to the hash group bitmap key technique and SQL. However, we can see that sometimes, and especially when looking for small itemsets, the SQL query is better than more complex methods. That is why an inductive database system should consider the different parameters of a subset query before executing it, so as to choose the best method. Notice that our method would be faster if it was implemented directly within the database instead of being working on top of a DBMS. We are now proceeding with several applications related to the field of bioinformatics. For instance, we have extracted a huge collection of a priori interesting sets of genes and we now collaborate with biologists to post-process them.

Acknowledgements. The authors thank Sophie Rome, Hubert Vidal and Jérémy Besson for the microarray data, the discretized matrix and the extracted closed itemsets.

6. REFERENCES

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[2] M. Botta, J.-F. Boulicaut, C. Masson, and R. Meo. A comparison between query languages for the extraction of association rules. In *Proc. DaWaK'02, LNCS 2454*, pages 1–10, 2002.

[3] J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Free-sets: a condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 7(1):5–22, 2003.

[4] L. De Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, January 2003.

[5] A. Hafez, J. S. Deogun, and V. V. Raghavan. The item-set tree: a data structure for data mining. In *Proc. DaWaK'99, LNCS 1676*, pages 183–192, 1999.

[6] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, Nov. 1996.

[7] R. Meo, G. Psaila, and S. Ceri. An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1998.

[8] T. Morzy and M. Zakrzewicz. Group bitmap index: a structure for association rules retrieval. In *Proc. ACM SIGKDD'98*, pages 284–288, 1998.

[9] J. Pei, J. Han, and R. Mao. CLOSET an efficient algorithm for mining frequent closed itemsets. In *Proc. SIGMOD Workshop DMKD'00*, 2000.

[10] R. Rantzau. Frequent itemset discovery with sql using a vertical layout and universal quantification. In *Proc. EDBT Workshop DTDM'02*, Praha, CZ, 2002.

[11] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Data Mining and Knowledge Discovery*, 4(2–3):89–125, 2000.

[12] A. Tuzhilin and G. Adomavicius. Handling very large numbers of association rules in the analysis of microarray data. In *Proc. SIGKDD'02*, 2002.

[13] A. Tuzhilin and B. Liu. Querying multiple sets of discovered rules. In *Proc. ACM SIGKDD'02*, 2002.

[14] M. J. Zaki. Generating non-redundant association rules. In *Proc. SIGKDD'00*, pages 34–43, 2000.