SQL Based Frequent Pattern Mining with FP-growth

Shang Xuequn, Sattler Kai-Uwe, and Geist Ingolf

Department of Computer Science
University of Magdeburg
P.O.BOX 4120, 39106 Magdeburg, Germany
{shang, kus, geist}@.iti.cs.uni-magdeburg.de

Abstract. Scalable data mining in large databases is one of today's real challenges to database research area. The integration of data mining with database systems is an essential component for any successful large-scale data mining application. A fundamental component in data mining tasks is finding frequent patterns in a given dataset. Most of the previous studies adopt an Apriori-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exist prolific patterns and/or long patterns. In this study we present an evaluation of SQL based frequent pattern mining with a novel frequent pattern growth (FP-growth) method, which is efficient and scalable for mining both long and short patterns without candidate generation. We examine some techniques to improve performance. In addition, we have made performance evaluation on DBMS with IBM DB2 UDB EEE V8.

1 Introduction

The integration of data mining with database systems is an emergent trend in database research and development area. This is particularly driven by explosion of the data amount stored in databases such as Data Warehouses during recent years, and database systems provide powerful mechanisms for accessing, filtering, indexing data and SQL parallelization. In addition, SQL-aware data mining systems have the ability to support ad-hoc mining, ie., allowing to mine arbitrary query results from multiple abstract layers of database systems or Data Warehouses. Mining frequent patterns in transaction databases has been studied popularly in data mining research. Most of the previous studies adopt an Apriori-like candidate set generation-and-test approach [3, 7, 8], which is based on an anti-monotone Apriori heuristic: if any length k pattern is not frequent in the database, its length (k+1) super-pattern can never be frequent. The above Apriori heuristic achieves good performance gain by reducing significantly the size of candidate sets. However, in situations with prolific frequent patterns, long patterns, or quite low minimum support thresholds, this kind of algorithm may still suffer from the following two nontrivial costs:

1. It is costly to handle a huge number of candidate sets.

2. It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.

Recently, an FP-tree based frequent pattern mining method [4], called FP-growth, developed by Han et al achieves high efficiency, compared with Apriori-like approach. The FP-growth method adopts the divide-and-conquer strategy, uses only two full I/O scans of the database, and avoids iterative candidate generation.

There are some SQL based approaches proposed to mine frequent patterns [9, 12], but they are on the base of Apriori-like approach. This fact motivated us to examine if we can get sufficient performance by the utilization of SQL based frequent pattern mining using FP-growth-like approach. We propose mining algorithms based on FP-growth to work on DBMS and compare the performance of these approaches using synthetic datasets.

The remainder of this paper is organized as follows: In section 2, we introduce the method of FP-tree construction and FP-growth algorithm. In section 3, we discuss different SQL based frequent pattern mining implementations using FP-growth-like approach. Experimental results of the performance test are given in section 4. We present related work in section 5 and finally conclude our study and point out some future research issues in section 6.

2 Frequent Pattern Tree and Frequent Pattern Growth Algorithm

The frequent Pattern mining problem can be formally defined as follows. Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items, and DB be a transaction database, where each transaction T is a set of items and $T \subseteq I$. An unique identifer, called its TID, is assigned with each transaction. A transaction T contains a pattern P, a set of items in I, if $P \subseteq T$. The support of a pattern P is the number of transactions containing P in DB. We say that P is a frequent pattern if $P^{'}$s support is no less than a predefined minimum support threshold ξ.

In [4], frequent pattern mining consists of two steps:

1. Construct a compact data structure, frequent pattern tree (FP-tree), which can store more information in less space.
2. Develop an FP-tree based pattern growth (FP-growth) method to uncover all frequent patterns recursively.

2.1 Construction of FP-tree

The construction of FP-tree requires two scans on transaction database. The first scan accumulates the support of each item and then selects items that satisfy minimum support. In fact, this procedure generates frequent-1 items and then stores them in frequency descending order. The second scan constructs FP-tree.

A FP-tree is a prefix-tree structure storing frequent patterns for the transaction database, where the support of each tree node is no less than a predefined

minimum support threshold ξ. Each node in the item prefix subtree consists of three fields: item-name, count, and node-link. Where node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none. For any frequent item a_i, all the possible frequent patterns that contain a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header. The frequent items in each path are stored in their frequency descending order. We give the following FP-tree construction algorithm introduced in [4].

Algorithm 1 (FP-tree construction)
Input: A transaction database DB and a minimum support threshold ξ.
Output: It's frequent pattern tree, FP-tree
Method: The FP-tree is constructed in the following steps.

1. Scan the transaction database DB once, get the frequent sets F and absolute frequencies. Sort F in frequency descending order as L, the list of frequent items.
2. Initialize the FP-tree T: T = node labelled "null". For each transaction in DB, remove infrequent items and sort the frequent ones according to the order of L. Let the sorted frequent item list in transaction be $[p \mid P]$, where p is the first element and P is the remaining list. Call insert-tree $(p \mid P, T)$ to add the resulting string to T, update counts as necessary.

Function insert-tree $(p \mid P, T)$
If T has a child N such that $N.item - name = p.item - name$
Then increment N's count by 1;
Else do {create a new node N;
N's count = 1;
N's parent link be linked to T;
N's node-link be linked to the nodes with the same item-name
via the node-link structure;}
If P is nonempty, call insert-tree (P, N).

Fig. 1. Algorithm for FP-tree construction

FP-tree is a highly compact and much smaller than its original database, and thus saves the costly database scans in the subsequent mining processes. Let's give an example with four transactions in Table 1, we get an FP-tree in Figure 2.

TID	$Transaction$	$(Ordered) Frequent Items$
1	$1, 3, 4$	$3, 1$
2	$2, 3, 5$	$2, 3, 5$
3	$1, 2, 3, 5$	$2, 3, 5, 1$
4	$2, 5$	$2, 5$

Table 1. A transaction database and $\xi = 2$

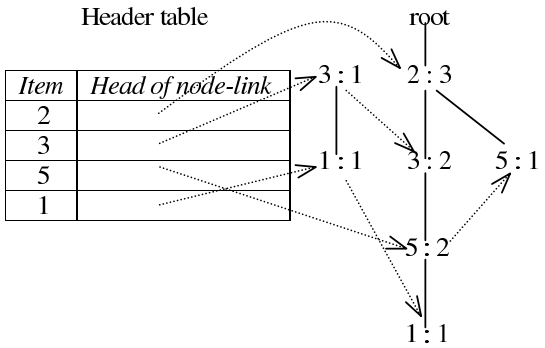

Fig. 2. An FP-tree for Table 1

2.2 FP-growth

Based on FP-tree structure, an efficient frequent pattern mining algorithm, FP-growth method is proposed, which is a divide-and-conquer methodology: decompose mining task into smaller ones, and only need sub-database test.

FP-growth performed as follows:

1. For each node in the FP-tree construct its conditional pattern base, which is a "subdatabase" constructed with the prefix subpath set co-occurring with the suffix pattern in the FP-tree. FP-growth traverses nodes in the FP-tree from the least frequent item in L;
2. Construct conditional FP-tree from each conditional pattern base;
3. Execute the frequent pattern mining recursively upon the conditional FP-tree. If the conditional FP-tree contains a single path, simply enumerate all the patterns.

With the FP-tree in Figure 2, the mining process and result is listed in Table 2.

$Item$	$ConditionalPatternBases$	$ConditionalFP-tree$	$FrequentPattern$
1	$\{(3:1),(2:1,3:1,5:1)\}$	$\langle 3:2\rangle$	3 1 : 2
5	$\{(2:2,3:2),(2:1)\}$	$\langle 2:3,3:2\rangle$	2 5 : 3, 3 5 : 2, 2 3 5 : 2
3	$\{(2:2)\}$	$\langle 2:2\rangle$	2 3 : 2
2	ϕ	ϕ	ϕ

Table 2. Mining of all-patterns based on FP-tree in Figure 2

3 Frequent Pattern Mining Based on SQL

Although an FP-tree is rather compact, it is unrealistic to construct a main memory-based FP-tree when the database is large. However using RDBMSs provides us the benefits of using their buffer management systems specifically developed for freeing the user applications from the size considerations of the data. And moreover, there are several potential advantages of building mining algorithms to work on RDBMSs. An interesting alternative is to store a FP-tree in a table. We studied two approaches in this category - FP, EFP (Expand Frequent Pattern). They are different in the construction of frequent pattern tree table, named FP. FP approach checks each frequent item whether it should be inserted into a table FP or not one by one to construct FP. EFP approach introduces a temporary table EFP, thus table FP can generate from EFP.

Transaction data, as the input, is transformed into a table T with two column attributes: transaction identifier (*tid*) and item identifier (*item*). For a given *tid*, typically there are multiple rows in the transaction table corresponding to different items in the transaction. The number of items per transaction is variable and unknown during table creation time.

FP-tree is a good compact tree structure. In addition, it has two good properties: node-link property (all the possible frequent patterns can be obtained by following each frequent's node-links) and prefix path property (to calculate the frequent patterns for a node a_i in a path, only the prefix sub-path of a_i in P need to be accumulated). For storing the tree in a RDBMS a flat table structure is necessary. According to the properties of FP-tree, we represent an FP-tree by a table FP with three column attributes: item identifier (*item*), the number of transactions that contain this item in a sub-path (*count*), and item prefix subtree (*path*). The field *path* is beneficial not only to construct the table FP but also to find all frequent patterns from FP. In the construction of table FP, the field *path* is an important condition to judge if an item in frequent item table F should be insert into the table FP or update the table FP by incrementing the item's count by 1. If an item does not exist in the table FP or there exist the same items as this item in the table FP but their corresponding *path* are different, insert the item into table FP. In the process of construction conditional pattern base for each frequent item, we only need to derive its all *path* in the table FP as a set of conditional paths, which co-occurs with it.

3.1 Construction of table FP

The process of the table FP construction is as following:

1. Transfer the transaction table T into table T', in which infrequent items are excluded and frequent ones are sorted in descending order by frequency, i.e. frequent 1-itemsets. SQL query using to generate T' from T in figure 3.
2. Construct the table FP. The algorithm for constructing the table FP is show in Figure 4.

```
insert into T'
select t.id, t.item from T t,
   ((select item, count(*) from T
   group by item
   having count(*) ≥ minsupp
   order by count(*) desc ) as F (item, count))
where t.item = F.item
order by t.id, F.count desc
```

Fig. 3. SQL query to generate T'

```
Algorithm 2 (table FP construction)
  Input: a transferred transaction table T'
  Output: a table FP
  Procedure:
  for items with the first identical tid in the table T'
    insert into the table FP
  for items with the other identical tids in the table T'
    insertFP (items);

  insertFP (items)
    count := 1;
    curpath := null;
    if FP has an item f = i1 (the first item in the items) and f. path = null
      for each item ik in the items
        insert ik into the table FP;
        curpath += ik;
    else
      for each item ik in the items
        if FP has not an item f = ik
          insert ik into the table FP;
        else
          if f. path != ik. path
            insert ik into the table FP;
          else
            curcount = ik. count + 1;
            update the table FP;
      curpath += ik;
```

Fig. 4. Algorithm for constructing table FP

3.2 Finding frequent pattern from FP

After the construction of a table FP, we can use this table to efficiently mine the complete set of frequent patterns. For each frequent item a_i we construct its conditional pattern base table $PB - a_i$, which has three column attributes $(tid, item, count)$. The table $PB - a_i$ includes items that co-occur with a_i in the table FP. As we said above, the path attribute in the table FP represent the information of prefix subpath set of each frequent itemset in a transaction. So this process is implemented by a select query to get all corresponding counts and paths, then split these paths into multiple items with the same count.

```
select count, path from FP where item = a_i;
for each count cnr, path p
  id := 1;
  item[ ] = split(p);
  for each item i in item[ ]
    insert into PB − a_i values (id, cnr, i);
  id += 1;
```

After then, we construct the table $ConFP - a_i$ from each conditional pattern base table $PB - a_i$ using the same algorithm as the table FP construction, and mine recursively in the table $ConFP - a_i$. The algorithm of finding frequent patterns from table FP is showed in Figure 5.

```
Algorithm 3 (FindFP)
  Input: table FP constructed based on Algorithm 2 and table F collects all
         frequent itemsets.
  Output: a table Pattern, which collects the complete set of frequent patterns.
  Procedure:
  If items in the table FP in a single path
    combine all the items and suffix, insert into Pattern;
  else
    for each item a_i in the table F
      construct table ConFP − a_i;
      if ConFP − a_i ≠ φ
        call FindFP(ConFP − a_i);
```

Fig. 5. Algorithm for finding frequent patterns from table FP

3.3 EFP approach

In the whole procedure, the construction of table FP (table $ConFP$) is a time-consuming procedure. The important reason is that each frequent item must be tested one by one to construct the table FP (table $ConFP$). In that case, the test process is inefficient.

From the above discussions, we expect significant performance improvement by introducing an extended FP table, called EFP, which has the same format as table FP $(item, count, path)$. We can obtain EFP by directly transforming frequent items in the transaction table T'. We initialize the path of the first frequent item a_1 in each transaction and set it as $null$. The path of the second frequent item a_2 is $null : a_1$, and the path of the third frequent item a_3 is $null : a_1 : a_2$, and so on. Table EFP represents all information of frequent itemsets and their prefix path in each transaction. We combine the items with identical path to get the table FP. However, compare to the construction of table FP, we do not need to test each frequent item to construct the table EFP and can make use of the database powerful query processing capability. For example, with the transactions in Table 1, we get a table FP and a table EFP in Figure 6.

Item	*Count*	*Path*
3	1	*null*
1	1	*null* : 3
2	3	*null*
3	2	*null* : 2
5	2	*null* : 2 : 3
1	1	*null* : 2 : 3 : 5
5	1	*null* : 2

(a) An FP table for Table 1

Item	*Count*	*Path*
3	1	*null*
1	1	*null* : 3
2	1	*null*
3	1	*null* : 2
5	1	*null* : 2 : 3
2	1	*null*
3	1	*null* : 2
5	1	*null* : 2 : 3
1	1	*null* : 2 : 3 : 5
2	1	*null*
5	1	*null* : 2

(b) An EFP table for Table 1

Fig. 6. table FP and table EFP for Table 1

3.4 Using SQL with object-relational extension

In the following section, we study approaches that use object-relational extension in SQL to improve performance. We consider an approach that use a table function path. As a matter of fact, all approaches above have to materialize its temporary table namely T' and PB'. Those temporary tables are only required in the construction of table FP and table $ConFP$. They are not needed for generating the frequent patterns. So we further use subquery instead of temporary tables. The data table T is scanned in the (id, item) order and combined with frequent itemsets table F to remove all infrequent items and sort in support descending order as F, and then passed to the user defined function Path, which collects all the prefixes of items in a transaction. SQL query to generate FP

using Path as follows:

insert into FP select $tt_2.item$, tt_2.count (*), $tt_2.path$
from (select $T.id$, $T.item$ from T, F
where $T.item = F.item$
order by $T.id$, $F.count$ desc) as $tt1$,
table (Path ($tt_1.id$, $tt_1.item$)) as tt_2
group by $tt_2.item$, $tt_2.path$
order by $tt_2.path$

4 Performance Evaluation

4.1 Dataset

We use synthetic transaction data generation with program describe in Apriori algorithm paper [3] for experiment. The nomenclature of these data sets is of the form TxxIyyDzzzK. Where xx denotes the average number of items present per transaction. yy denotes the average support of each item in the data set and zzzK denotes the total number of transactions in K (1000's). We report experimental results on four data sets, they are respectively T5I5D1K, T5I5D10K, T25I10D10K, T25I20D100K.

4.2 Performance Comparison

Our experiments were performed on DBMS with IBM DB2 UDB EEE V8. For comparison, we also implemented a loose-coupling approach, in which access to data in DBMS was provided through a JDBC interface, then construct an FP-tree in memory and a k-way join approach, that proposed in [12]. We built a (id, item) index on the data table T and a (item) Index on the frequent itemsets table F. The goal was to let the optimizer choose the best plan possible. In figure 7 (a)(b) we show the total time taken by the four approaches on data set T5I5D1K and T5I5D10K: k-way join approach, loose-coupling approach, SQL-based FP, and improved SQL-based EFP (without object-relational extension in SQL). From the graph we can make the following observation: k-way join, FP, EFP has the better performance than loose-coupling approach. EFP approach can get competitive performance out of FP implementation. An important reason for superior performance of EFP over FP is the avoid testing each frequent item one by one in the construction of table FP. For instance, for dataset T5I5D10K with the support value of 0.5%, in the FP approach, almost 50% of execution time belongs to the construction of table FP. However, in the EFP approach, almost less 24% of execution time belongs to the construction of table FP. Since the recursive construction of table $ConFP$ use the same method as the construction of table FP. In that case, the overall execution time is highly reduced.

We compare the four approaches on data sets T25I10D10K and T25I20D100K: k-way join approach, loose-couple approach, EFP approach, and Path approach

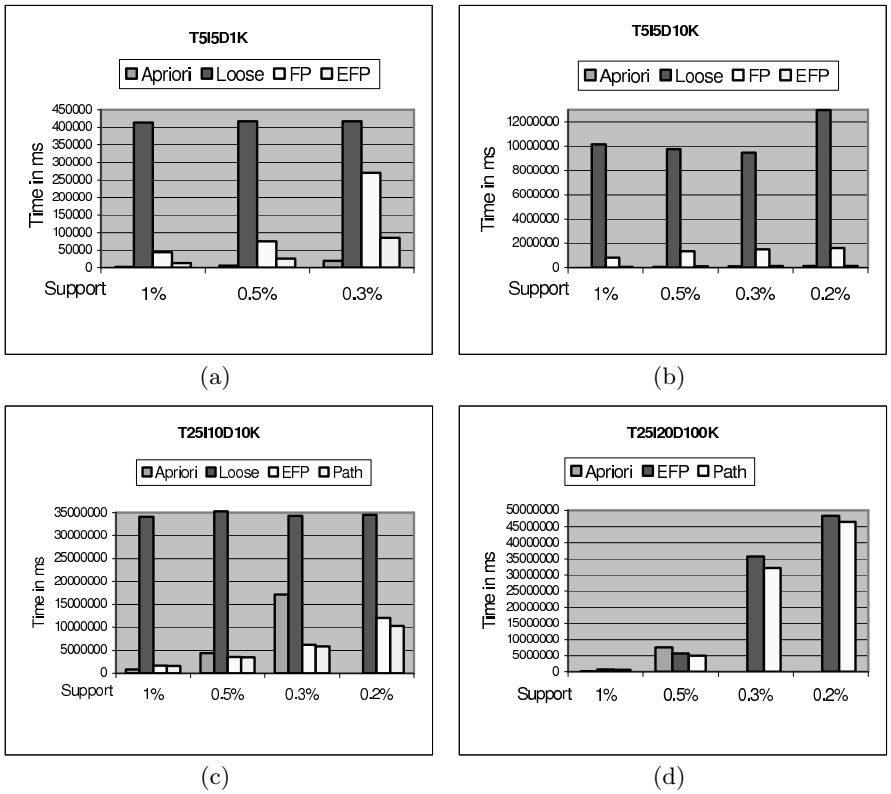

Fig. 7. Comparison of four approaches. In (c), for k-way join approach with the support value of 0.2% , and in (d), for Loose and k-way join approach with the support value of 0.3% and 0.2%, the running times were so large that we had to abort the runs in many cases.

using a user defined table functions (Path). Figure 7 (c)(d) shows the results of experiments. From the graph we can make the following observation: EFP and Path approach can get better performance than k-way join on large data sets or long patterns. The main reason is that generating candidate-k table C_k is time-consuming procedure when T is large or minimum support threshold is quite low. In addition, with the extended SQL we can get the improved performance.

5 Related Work

The work on frequent pattern mining started with the development of the AIS algorithm, and was further modified and extended in [3]. Since then, there have been several attempts in improving the performance of these algorithms. [7] presents a hash based algorithm, which is especially effective for the generation of candidate set for large2-itemsets. [8] presents a partition algorithm, which improve the overall performance by reducing the number of passes needed over the complete database to at most two. [1] presents a TreeProjection method, which represents frequent patterns as nodes of a lexicographic tree and uses the hierarchical structure of the lexicographic tree to successively project transactions and uses matrix counting on the reduced set of transactions for finding frequent patterns. [4] builds a special tree structure in main memory to avoid multiple scans of database. However, most of these algorithms are applicable to data stored in flat files. The basic characteristics of these algorithms are that they are main memory algorithms, where the data is either read directly from flat files or is first extracted from the DBMS and then processed in main memory.

Recently researchers have started to focus on issues related to integrating mining with databases. There have been language proposals to extend SQL to support mining operators. The data mining Query Language DMQL [5] proposed a collection of such operators for classification rules, characteristics rule, association rules, etc. The Mine Rule operator [6] was proposed for a generalized version of the association rule discover problem. [2] presents a methodology for tightly-coupled integration of data mining applications with a relational database system. [9] has tried to highlight the implications of various architecture alternatives for coupling data mining with relational database systems. They have also compared the performance of the SQL-92 architecture with SQL-OR based architecture, and they are on the base of Apriori-like approach.

6 Summary and Conclusion

We have implemented SQL based frequent pattern mining using FP-growth-like approach. We represent FP-tree using a relational table FP and proposed a method to construct this table. To improve its performance, a table called EFP is introduced, which is in fact stores all information of frequent item sets and their prefix path in each transaction. And then, table FP can derived from table EFP. Compare to the construction of FP, the process of the construction of EFP avoid testing each frequent item one by one. We next experimented with

an approach that made use of the object-relational extensions like table function. The experimental results show that SQL based frequent pattern mining approach using FP-growth can get better performance than Apriori on large data sets or long patterns.

There remain lots of further investigations. We plan to implement our SQL based frequent pattern mining approach on parallel RDBMS, and to check how efficiently our approach can be parallelized and seeded up using parallel database system. Additionally, is there an SQL based algorithm which combine Apriori and FP-growth to scale both small and large data sets?

References

1. R. Agarwal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing(Special Issue on High Performance Data Mining), 2000.
2. R. Agrawal and K. Shim. Developing tightly-coupled data mining application on a relational database system. In Proc.of the 2nd Int. Conf. on Knowledge Discovery in Database and Data Mining, Portland,Oregon, 1996.
3. R. Agrawal, R. Srikant. Fast algorithms for mining association rules. In Proc. of the 20st VLDB Conference, Santiago, Chile, pp.487-499, 1994.
4. J. Han, J. pei, and Y. Yin. Mining frequent patterns without candidate generation. In Proc. of the ACM SIGMOD Conference on Management of data, 2000.
5. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. DMQL: A data mining query language for relational database. In Proc. Of the 1996 SIGMOD workshop on research issues on data mining and knowledge discovery, Montreal, Canada, 1996.
6. M. Houtsma and A. Swami. Set-oriented data mining in relational databases. DKE, 17(3): 245-262, December 1995.
7. R. Meo, G. Psaila, and S. Ceri. A new SQL like operator for mining association rules. In Proc. Of the 22nd Int. Conf. on Very Large Databases, Bombay, India, 1996.
8. J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In Proc. of the ACM SIGMOD Conference on Management of data, pp.175-186, 1995.
9. I. Pramudiono, T. Shintani, T. Tamura and M. Kitsuregawa. Parallel SQL based associaton rule mining on large scale PC cluster: performance comparision with directly coded C implementation. In Proc. Of Third Pacific-Asia Conf. on Knowledge Discovery and Data Mining, 1999.
10. R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. DMKD03: 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2003.
11. A. Savsere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Proc. of the 21st VLDB Conference, 1995.
12. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating mining with relational database systems: alternatives and implications. In Proc. of the ACM SIGMOD Conference on Management of data, Seattle,Washinton,USA, 1998.
13. K. Sattel and O. Dunemann. SQL database primitives for decision tree classifiers. In Proc. Of the 10nd ACM CIKN Int. Conf. on Information and Knowledge Management, Atlanta,Georgia, 2001.

14. S. Thomas and S. Chakravarthy. Performance evaluation and optimization of join queries for association rule mining. In Proc. DaWaK, Florence, Italy, 1999.
15. H. Wang and C. Zaniolo. Using SQL to build new aggregates and extenders for Object-Relational systems. In Proc. Of the 26th Int. Conf. on Very Large Databases, Cairo,Egypt, 2000.
16. T. Yoshizawa, I. Pramudiono, and M. Kitsuregawa. SQL based association rule mining using commercial RDBMS (IBM DB2 UDB EEE). In Proc. DaWaK, London, UK, 2000.