Shaping SQL-Based Frequent Pattern Mining Algorithms

Csaba István Sidló[1] and András Lukács[2]

[1] Eötvös Loránd University, Faculty of Informatics,
Pázmány Péter sétány 1/c, 1117 Budapest, Hungary
scs@elte.hu
[2] Computer and Automation Research Institute, Hungarian Academy of Sciences,
Kende u. 13-17., 1111 Budapest, Hungary
alukacs@sztaki.hu
WWW home page: http://informatika.ilab.sztaki.hu/websearch/

Abstract. Integration of data mining and database management systems could significantly ease the process of knowledge discovery in large databases. We consider implementations of frequent itemset mining algorithms, in particular pattern-growth algorithms similar to the top-down FP-growth variations, tightly coupled to relational database management systems. Our implementations remain within the confines of the conventional relational database facilities like tables, indices, and SQL operations. We compare our algorithm to the most promising previously proposed SQL-based FIM algorithm. Experiments show that our method performs better in many cases, but still has severe limitations compared to the traditional stand-alone pattern-growth method implementations. We identify the bottlenecks of our SQL-based pattern-growth methods and investigate the applicability of tightly coupled algorithms in practice.

1 Introduction

Frequent itemset mining (FIM) is a central exercise of data mining. FIM is a base to solve several further tasks like association rule, sequential and other frequent pattern mining. Although algorithms for FIM were studied exhaustively (see e.g. [1]), much fewer results and solutions are known about FIM algorithms implemented in and for relational database management systems. On the other hand the demand for integration of data mining tools into the existing database management systems is tangible. An obvious next step solution is the extension of the existing database query languages with new functions supporting FIM algorithms.

Comparing the SQL-based implementations to the stand-alone FIM algorithms one can notice that the second class contains the very well performing pattern-growth algorithms [10, 19], while the idea of pattern-growth is poorly represented among the available SQL-based FIM algorithms [25]. The main proposal of this paper is to eliminate this flaw by suggesting a new pattern-growth

FIM algorithm tightly coupled to relational database management systems. Therefore we examine SQL-based FP-growth algorithms in a performance perspective, that is whether they are usable in practice. The main result is the efficient implementation of the sophisticated FP-growth algorithm. We expect that our algorithms do the data processing inside the database. We identify the bottlenecks of the algorithms in order to determine the promising directions for development of data mining enabled database systems.

2 Integrating Data Mining and Databases, Related Work

Data mining addresses extraction of interesting knowledge from large databases. However, this complements the goals of the data warehouse and on-line analytical processing technologies, the chasm between the existing *data mining* and the *database* world is rather wide. Most data mining solutions include fully database-independent applications for the data mining tasks. We belive that coupling data mining with relational databases would remarkably improve the efficiency in the knowledge discovery process and simplify the construction of decision support systems.

Inductive databases [12, 7] are databases that integrate data with knowledge. The main goal of an inductive database is to allow the user not only to query the data that resides in the database, but also to query and mine generalizations, patterns of interest. The knowledge discovery process should be supported by an integrated framework, the user should be allowed to perform different operations on both data and patterns. The interaction takes place through inductive query languages supporting data mining, which are often extensions of SQL. A good comparison between languages supporting descriptive rule mining can be found in [6]. Other directions allowing data mining-like queries are data mining query interfaces and APIs [18]. From the analyst point of view the usability of OLAP (on-line analytical processing) systems could also be significantly increased by the integration of data mining methods. This viewpoint of inductive databases is the on-line analytical mining (OLAM) [9].

Despite the probable usefulness we are still far away from a general theory and practical realizations of full value of inductive databases, however, there are promising partial results, and also RDBMS vendors try to integrate more and more knowledge discovery support in their systems, turning them into decision support platforms (see [14] and [15]).

The accomplishment of integration from the architectural point of view is still an open question. In case of the fully separated systems, the required data is read from the DBMS, the mining is performed on a file system-cached version, and the results are written back to the database. The advantage here is the possibility to use special memory structures and buffer strategies. The loosely coupled architectures access the data through some standard interface too, but push parts of the data mining tasks in the DBMS. The tightly coupled variants use only facilities of the DBMS. A tightly coupled architecture is introduced in [16].

Nonetheless, SQL-based tightly coupled algorithms are considered bearing significantly inferior in terms of running times compared to stand-alone implementations, there exist advantages of tightly coupled data mining. Since data appears mostly in data warehouses and other databases in practice, in the case of tightly coupled data mining applications no additional data mining system is needed. Databases have already solved the problem of efficient and safe storing and querying large datasets reliably. Therefore, DBMSs can facilitate data mining to become an online, robust, scalable and concurrent process by complementing the existing querying and analytical functions. A relevant example is that of the caching problem. When data structures are too large to fit in memory, we can try to entrust caching to the database engine.

The first attempt to the particular problem of integrated frequent itemset mining was the SETM algorithm [11], expressed as SQL queries working on relational tables. The Apriori algorithm [2] opened up new prospects for FIM. The database-coupled variations of the Apriori algorithm were carefully examined in [22]. The SQL-92 based implementations were too slow, but the SQL implementations enhanced with object-relational extensions (SQL-OR) performed acceptable. The so-called Cache-Mine implementation had the best overall performance, where the database-independent mining algorithm cached the relevant data in a local disk cache. The optimization of the key operation, the join queries was studied in [26], and a new SQL-92-based method, Set-oriented Apriori was introduced. Further performance evaluations on commercial RDBMS can be found in [28], evaluations of the SQL-OR option in [17]. An interesting SQL-92 algorithm based on universal quantification is discussed in [20] and [21].

Since the introduction of the FP-growth method [10], a few attempts were made to implement pattern-growth methods inside the RDBMS [25]. [3] presents a novel, FP-tree-based indexing method, which provides a complete and compact representation of the dataset for frequent itemset mining, and collaborates efficiently with the relational database kernel. [8] deals with database-independent frequent itemset mining from secondary memory.

FIM is investigated most intensively among the problems of data mining in DBMS, but other classical data mining tasks are also studied, e.g. building and applying decision tree classifiers [23, 5].

3 Association Rule and Frequent Itemset Mining

Several data mining tasks, including identification of joint distribution, compression, and fast counting can be reduced to association rules mining.

Let us consider the set of *items* $I = \{item_1, item_2, \ldots item_m\}$. A setsystem $D \subseteq P(I)$ of I is called *database* and the elements of D are the *baskets* of items. The *support* of an *itemset* $A \subset I$ is the number of baskets that have all of the items from A. We call an itemset A *frequent* if A has a support greater than some fix threshold s. Finding all frequent itemsets is the goal of the *frequent itemset mining* (FIM).

Association rules are binary relations between itemsets. An *association rule* $A \rightarrow B$ is an ordered pair of two disjoint itemset, here A and B. The support of the rule $A \rightarrow B$ is the support of $A \cup B$, the number of baskets containing $A \cup B$. The confidence of this rule is defined by the ratio of the support of the set $A \cup B$ to the support of the set A. The aim of association rule mining is to find all the rules that have a support and confidence greater than or equal to some previously given s and c, respectively. Practically association rule mining can be derived to the frequent itemsets mining problem.

To solve the FIM problem one can observe that frequent itemsets satisfy the *antimonotonicity property* (or Apriori principle), for a subset A of itemset B the support of A is greater or equal to the support of B. This property is the base of the multi-pass algorithm called *Apriori* [2]. Further algorithms solving the FIM problem are based on pattern-growth [10, 19].

4 Apriori-Based Methods

The Apriori algorithm iterates two basic phases to find frequent itemsets. In the nth iteration step it generates at first candidates for frequent itemsets having size n, which can be done utilizing the Apriori principle: the nth candidate set C_n can be produced from the $(n-1)$th set of frequent itemsets F_{n-1}. Next it tests the candidate set against the database, by counting support values for the candidates. The process iterates until the candidate itemset becomes empty. We don not have to materialize C_1, all items in the database are candidates, and in all other cases we materialize C_n and F_n. Next we discuss the SQL-92 methods briefly.

The SQL implementations differ in data representation. Two basic variations to represent these sets in relations are the horizontal approach, where C_n and F_n have the schema $(item_1, item_2, ...item_n)$, and the vertical approach with $(set\_id, item)$ schema. The horizontal approach have the disadvantage that the item count is limited by the possible count of table attributes, but can be beneficial in the performance view. The input database table has always $(transaction\_id, item)$ schema, because of the unknown number of items per transaction, and fits mostly to the star schema in relational data warehouses.

The implementations also differ in the SQL commands for candidate generation and support counting. Since the support counting phase is the most time consuming part of the computing, most algorithms share the candidate generation operation, using a k-way join to generate C_n from F_{n-1}. The support counting commands like K-Way-Join, Subquery and 2-Way-Join utilize join operations, or rely on group by computations like Two-Group-Bys [22]. The basic K-Way-Join support counting joins the data table n times in the nth step:

insert into F_n select $item_1$ ... $item_n$ $count(*)$
from C_n, F_{n-1} as I_1, ... F_{n-1} as I_n
where $I_1.item < C_n.item_1$ and ... and $I_n.item < C_n.item_n$ and
$I_1.tid = I_2.tid$ and ... and $I_{n-1}.tid = I_n.tid$
group by $item_1$, ... $item_n$

having $count(*) \geq minsup$.

Subquery is an optimization of the K-Way-Join, which makes use of the common prefixes between the itemsets in the candidate set. We developed different versions of Subquery to apply the divide-and-conquer idea of [24]: if we divide the database into distinct partitions, then an itemset can only be frequent, if it is frequent on at least one partition. It is possible therefore to partition the input table, find the frequent itemsets over the partitions, then test all partition-wise valid frequent itemsets over the whole input table. Unfortunately, as depicted on Figure 3, the execution times against the size of the input table don not allow to efficiently apply the partition trick. However, the method could be used to mine data stored on multiple databases, as shown in [13].

5 Pattern-Growth Methods

Pattern-growth methods, first published in [10], represent the database in a compact data structure, called Frequent-Pattern-tree (FP-tree) to avoid repeated database scans and generating large candidate sets. The FP-tree stores items have greater support than the minimum support in a tree structure. Given an ordering of the items, transactions are represented as paths from the root node, sharing the same upper path if their first few frequent items are the same. The FP-tree is searched recursively to find the frequent itemsets with the FP-growth method.

Figure 1 shows an FP-tree built for an example database with minimum support threshold 2. All node is labelled by an item, has a count value and a sidelink to its siblings. The count value refers to the support of the path's itemset from the root to the given node. An additional header table stores the initial sidelink and the total item count for the items.

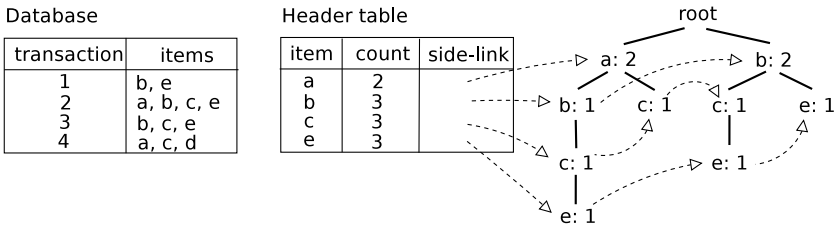


Fig. 1. FP-tree for a given database, built with minimum support threshold 2

5.1 Constructing the FP-tree

We represent the FP-tree in a natural fashion in a table having the schema

$$node : (node\_id, parent\_id, item, count, sidelink).$$

In a particular state of processing the binary sidelink attribute shows, whether a node is part of the processed subtree or not (Y/N). On the first level of the tree the parent attributes are null. An alternative approach can be found in [25], where instead of the parent reference, a specially implemented and not fully discussed path attribute is used for all nodes.

The FP-tree can be built by reading the database once, inserting a new path per transaction into the tree if the itemset of its frequent items has not been represented yet, or increasing the counts else. This method expressed as SQL queries is, however, not efficient, because of the cost of the *node* table accesses individually for all items. Instead of that, we build the FP-tree level by level, inserting all nodes on a particular level of the tree by one SQL query.

The first version we present uses the subset of the original input table containing only the transaction parts of frequent items, and a table containing ($node\_id$, $item$) elements, representing the prefix we have processed. We delete the processed rows from the filtered input table, get the next item per transaction by a minimum search, and insert new rows in *node*. Supposing that input table is $tdb\_filtered : (tid, item)$, the prefix table is $prefix : (tid, node\_id)$, and $node\_seq.nextval$ is used to generate the unique identifiers, the key step is:

```
insert into node
    select    node_seq.nextval, min.minitem,
              prefix.node_id, count(min.tid)
    from      ( select    tid, min(item) minitem
                from      tdb_filtered
                group by  tid ) min, prefix
    where     min.tid = prefix.tid
    group by  min.item, prefix.node_id.
```

Our second version uses an analytic function called *dense rank* to produce a sorted and filtered version of the input table. We create groups with the help of this function, according to the *tid* attribute, and rank the items in the group based on the given ordering (supposing that tdb:$(tid, item)$ is the input table):

```
select tid, item, dense_rank() over ( partition by tid order by item ) rank
from   tdb.
```

In this case the filtered input table is $tdb\_filtered : (tid, item, rank)$. Building *node* is similar to the previous version, but we can eliminate the minimum search and the deleting by referring to all levels by the rank value.

Items in the input table are represented by identifiers, and a natural ordering is given for them, but this ordering is not suitable for building the FP-tree. We use an additional table for the items, in which they are ordered according to exactly one new identifier. The new identifiers are given so, that the natural ordering of them will be the same as the descending ordering of the original items based on the count of transactions they appear in. This ordering promises optimal tree structure in the sense of compactness. This step can be solved by a simple sorting query, and the results can be used initially to fill up the header table described later.

5.2 FP-tree Evaluation

To avoid the combinatorial problem of evaluating the FP-tree, we use a method similar to the top-down FP-growth described in [27], which enables finding all frequent itemsets without materializing conditional subtrees. The core of the algorithm is a recursive procedure utilizing SQL operations and additional tables. The $header : (header\_id, item, count)$ table stores count information for items coming up in stages of the recursion, and also serves as a recursion heap. Header table identifiers are analogous to the separate header tables in the original FP-growth concept. All those itemsets are considered in a recursion step, which end up with a given $\bar{x}$ item sequence. An other table $header\_postfix : (header\_id, item)$ stores these $\bar{x}$ postfixes for the header identifiers. The **mine** procedure recursively produces all frequent itemsets above a given *minsup* minimal support value. First after the FP-tree creation phase it is called **mine(0)**, when *header* is already filled with frequent items and their counts, and rows refer to the initial *header_id* **0**.

Procedure mine(h_id)
1 **for** h_rec in (select $header\_id$, $item$, $count$ from $header$
  where $header\_id = h\_id$)
2 **if** $h\_rec.count \geq minsup$ **then**
3 output long pattern: ($h\_rec.item$, postfix) using $header\_postfix$;
4 $new\_header \leftarrow$ generate new header id ;
5 **for** each n node from $node$ located on paths
  upwards from $h\_rec.item$-s, having $sidelink = Y$
6 $n.count \leftarrow$ sum of counts of leaves ;
7 $n.sidelink \leftarrow Y$;
8 **if** (new_header, $n.item$) exists in $header$ **then**
9 add $n.count$ to header row identified by (new_header, $n.item$)
10 **else** insert $(new\_header, n.item, n.count)$ into $header$;
11 **for** each n node from $node$ not located on paths upwards from
  $h\_rec.item$-s, having $sidelink = Y$ and $item < h\_rec.item$
12 $n.sidelink \leftarrow N$;
13 **mine**(new_header) **;**

We implemented the steps of the algorithms as SQL queries, with the help of auxiliary tables. Frequent sets are outputted to the $result : (set\_id, item)$, absolute support values of itemsets to the $result\_support : (set\_id, support)$ table.

The main observation that motivated the top-down FP-growth method was, that if we process the tree in a top-down fashion, then the counts of the nodes above the actual leaf are no longer needed, therefore they can be used for counting. We use further temporary tables for the purpose of climbing up the paths and setting sidelinks and counts (rows 5-12). Table $path : (node\_id, count)$ stores the nodes found on the paths with the actual *count* value. We climb up the paths level by level, accumulating the counts of the leaves. The required information (original node, actual node, count value of the original node) for

these steps are stored in an other auxiliary table. This step can be solved by the use of a recursive query as well (assuming the syntax of Oracle):

select *node_id* from *node*
start with *node_id* = (actual node) connect by prior *parent_id* = *node_id*.

Processing the nodes on the paths leaf by leaf with the use of a recursive query instead of level by level, however, was less effective according to our early test results.

5.3 Indices and Further Optimization

It became clear after implementing the first versions of the algorithm, that the main cost arises from the *node* table accesses, especially from updates (steps 6, 7 and 12). These accesses refer to more and more node by the end of the processing, when we process nodes near to the leaves. We can optimize the updates, for example updating only those sidelinks of the nodes which do not have the right value yet, but after all without the use of indices these steps require full scans of the *node* table, and this costs mostly lots of block reads and writes.

The *node_id*, *parent_id* and *item* attribute values don't change after building the *node* table. It is therefore profitable to use standard B-tree indices on them, like $(item, node\_id)$ for searching *node_id*-s by *item*, or $(node\_id, parent\_id)$ to find parent nodes efficiently. The *sidelink* and *count* values are changed frequently. We don't want to access the table by the *count* attribute, but the use of some index on the *sidelink* attribute can be profitable. We can use regular indices, or, since the *sidelink* attribute has only two distinct attributes, we can use bitmap index. We tested both the regular and bitmap version for *sidelink*. Bitmap indices are good for selecting rows by low cardinality attributes, and the access times were really lower in practice. We refer hereafter the indexed version of the above described algorithm as FP-TDG.

We implemented several alternatives of FP-TDG. We experienced, that denormalizing the *node* table is beneficial: the *node*:$(node\_id, parent\_id, item)$, *sidelink*:$(node\_id)$, *count*: $(node\_id, count)$ schema enables us to manage the frequently changed information apart from the permanent tree-structure information. In this case we store the binary "header" information as a *node_id* set. The "count" values for nodes are stored in a smaller and exclusive table. Instead of building separate indices on these two tables we store them as B-trees with the help of the so-called "index-organized table" facility of the database server. We refer this version as FP-TDG2.

6 Experiments

Our experiments were performed on Oracle 9i Enterprise Edition release 2, installed on a PC server with two 3 GHz Intel Pentium processor, 2 GB memory, RAID-5 with IDE disks and Debian Linux operating system. The memory usage

of the database server was limited to 1 GB, because of other background services on the server. Redo logging was reduced for all tables, and parallel processing functions of the database were not enabled.

We generate the sequences of SQL operations by PL/SQL procedures. PL/SQL could be exchanged to any other programming language, which can connect to the database server through some standard database API. The code can be executed on an arbitrary client, because generating the SQL statements requires only little computing and networking capacity.

We used public FIMI [1] datasets as test datasets (table 1 shows the properties of the four selected datasets for demonstration).

Table 1. Dataset properties

Dataset	Num. of records (K)	Num. of transactions	Num. of items	Avg. num. of items per transaction
RETAIL	908.576	88,162	16,469	10.3
T10I4D100K	250	100,000	250,000	10
ACCIDENTS	11,498	340,183	468	33.8
BMS-WEBVIEW-2	358.278	77,512	3,340	4.6

We have chosen the Subquery method to compare our algorithms to, because - as suggested in [22] - Subquery had the best overall performance (although this is in fact opposed to the result in [21], where K-Way-Join is superior in this category). We implemented our version with the so-called second-pass optimization: we don not materialize the candidates of size two, we replace it with a 2-way join between frequent item tables of size one.

The other algorithm we have chosen to compare is the *nonordfp* algorithm [1], which is a fully database-independent implementation, coded in C++. The algorithm *nonordfp* handles an FP-tree-like structure, and can efficiently evaluate it without materializing subtrees. We implemented a tiny cache-mine system, where the application runs on the hardware of the database, but only connects to the database to read out the input data and to write back the result through standard JDBC interface. The code of *nonordfp* caches the data in the filesystem for processing. The memory usage was not limited.

The main part of *nonordfp*'s total execution times was the time needed to read and write the database. The response time goes up only below really low minimum support values, when the result set becomes large. The algorithm *nonordfp* outperforms the SQL-based methods for low minimum support, however as being in-memory algorithm, the input size is limited by the available memory.

Figure 2 shows the execution times of our two versions of creating the FP-tree. Figure 3 (a.) shows execution times with different sized samples of the RETAIL database at the minimum support value of 0.5 %. Figures 3 (b.) and 4 compare the total execution times of our algorithms. FP-TDG and FP-TDG2 mostly outperform Subquery, but in case of the T10I4D100K generated dataset they do not perform well. This dataset is rather sparse, and most FP-growth

methods work less efficiently on sparse datasets. This can be seen here as well. The FP-tree becomes too large, it does not compress the database efficiently, and this causes a leap in the aggregated node-access times. On the other hand the sparsity of the database is advantageous for the join-based Apriori methods, when the size of the candidate sets shrink fast.

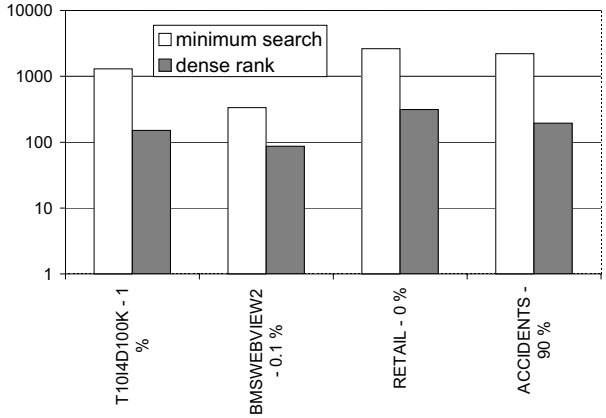
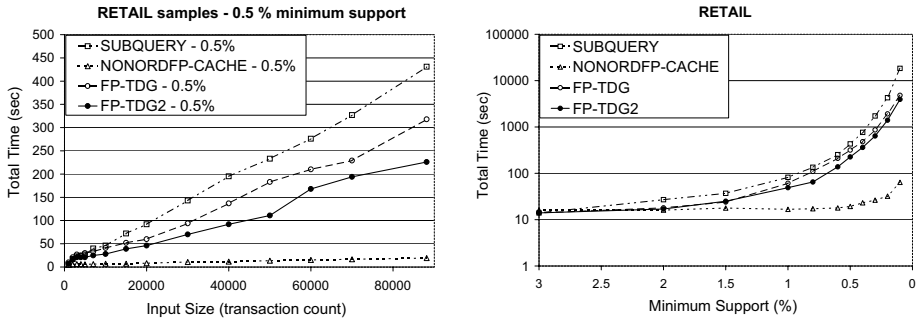


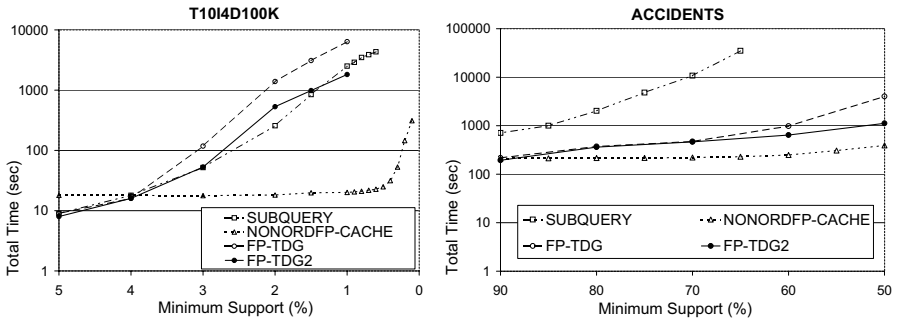
Fig. 2. Constructing the FP-tree for some selected databases and minimum support values



(a.) (b.)

Fig. 3. Execution times for the RETAIL dataset

We have tested Subquery and FP-TDG2 in real-life environment, over logs of the largest Hungarian web portal [origo] (www.origo.hu). The site produces 7,000,000 page hits on a typical workday, which is processed by an experimental weblog mining architecture (see [4] for details). The preprocessed data is stored



(a.) (b.)

Fig. 4. Execution times for the T10I4D100K and the ACCIDENTS datasets

in the central Oracle 9i database component of the architecture. The task is to identify frequent pages accessed together by the users on a given day. Execution times of a typical workday can be seen on Figure 5, where 767,663 identified user accessed 57,911 different pages in the course of the day, which resulted in 2,395,146 records. The average number of pages per user was 3.12.

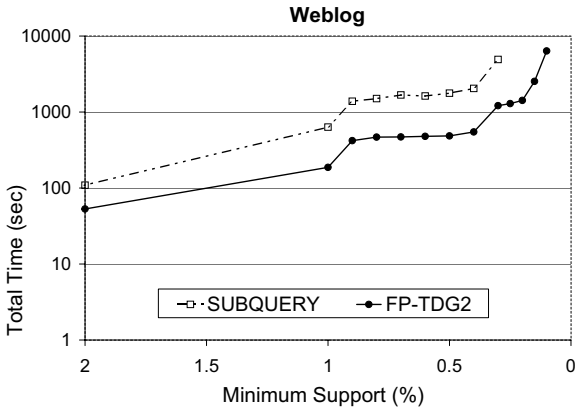


Fig. 5. Execution times on weblog data

The results can be analysed in our architecture by the given statististical analysis framework, through a webserver with dynamic web pages, connected to the database. Users can discover frequent page sets by extending these sets one-by-one, starting with an empty set, or with a directly given set. They can see on every page the details of the given page set, and the toplist of the next element extending this set. This simple method is suitable in our case, where (for

the minimum support interval of Figure 5) we have 1 to 47 thousand frequent sets with a maximum size of 13.

In this practical use of the SQL-based FP-TDG2 we eliminated the need for a separate FIM system with duplicated data. Frequent sets are produced with only the use of the common database facilities. The execution times are acceptable, they are comparable to the computation times of some complex statistical aggregations in the database.

Implementations of the algorithms and the used sample datasets can be reached at http://scs.web.elte.hu/sqlfim/.

7 Conclusion

In this paper, we proposed an FP-tree based algorithm for frequent itemset mining, discussed variants for both constructing and evaluating the tree. Our final algorithm performed well in its category, but it has severe limitations in performance compared to stand-alone FIM algorithms. We found that the common buffer management and indexing technics do not provide enough support for the task of efficient storing and accessing the FP-tree in relations, which is the essential problem of our SQL-based method. However, practical uses of our algorithm seems possible, especially for high support thresholds, where the result set still has a manageable size.

References

1. Frequent itemset mining implementations repository. http://fimi.cs.helsinki.fi/.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
3. E. Baralis, T. Cerquitelli, and S. Chiusano. Index support for frequent itemset mining in a relational DBMS. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 754–765. IEEE Computer Society, 2005.
4. A. A. Benczúr, K. Csalogány, K. Hum, A. Lukács, B. Rácz, C. Sidló, and M. Uher. Architecture for mining massive web logs with experiments. In *Proceedings of the HUBUSKA Open Workshop on Generic Issues of Knowledge Technologies*, 2005.
5. F. Bentayeb and J. Darmont. Decision tree modeling with relational views. In *ISMIS '02: Proceedings of the 13th International Symposium on Foundations of Intelligent Systems*, pages 423–431. Springer-Verlag, 2002.
6. M. Botta, J.-F. Boulicaut, C. Masson, and R. Meo. Query languages supporting descriptive rule mining: A comparative study. In *Database Support for Data Mining Applications*, volume 2682/2004 of *Lecture Notes in Computer Science*, pages 24–51. Springer-Verlag, 2004.
7. J.-F. Boulicaut, M. Klemettinen, and H. Mannila. Modeling KDD processes within the inductive database framework. In *DaWaK '99: Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, pages 293–302. Springer-Verlag, 1999.

8. G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. In *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 91–98, Washington, DC, USA, 2004. IEEE Computer Society.
9. J. Han. Towards on-line analytical mining in large databases. *SIGMOD Rec.*, 27(1):97–107, 1998.
10. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.
11. M. Houtsma and A. Swami. Set-oriented data mining in relational databases. *Data Knowl. Eng.*, 17(3):245–262, 1995.
12. T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Commun. ACM*, 39(11):58–64, 1996.
13. H. Kona and S. Chakravarthy. Partitioned approach to association rule mining over multiple databases. pages 320–330, 2004.
14. W. Li and A. Mozes. Computing frequent itemsets inside Oracle 10g. In *VLDB'04*, pages 1253–1256, 2004.
15. J. MacLennan. SQL Server 2005: Unearth the new data mining features of analysis services 2005. *MSDN Magazine*, 19(9), 2004.
16. R. Meo, G. Psaila, and S. Ceri. A tightly-coupled architecture for data mining. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 316–323, Washington, DC, USA, 1998. IEEE Computer Society.
17. P. Mishra and S. Chakravarthy. Performance evaluation of SQL-OR variants for association rule mining. *Lecture Notes in Computer Science*, 2737/2003:288–298, 2003.
18. A. Netz, S. Chaudhuri, U. M. Fayyad, and J. Bernhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *Proceedings of the 17th International Conference on Data Engineering*, pages 379–387. IEEE Computer Society, 2001.
19. J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 441–448. IEEE Computer Society, 2001.
20. R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 20–27. ACM Press, 2003.
21. R. Rantzau. Frequent itemset discovery with SQL using universal quantification. In *Database Support for Data Mining Applications*, pages 194–213, 2004.
22. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 343–354. ACM Press, 1998.
23. K.-U. Sattler and O. Dunemann. SQL database primitives for decision tree classifiers. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 379–386. ACM Press, 2001.
24. A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 432–444. Morgan Kaufmann Publishers Inc., 1995.
25. X. Shang, K.-U. Sattler, and I. Geist. SQL based frequent pattern mining with fp-growth. In *INAP/WLP*, pages 32–46, 2004.

26. S. Thomas and S. Chakravarthy. Performance evaluation and optimization of join queries for association rule mining. In *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, pages 241–250. Springer-Verlag, 1999.
27. K. Wang, L. Tang, J. Han, and J. Liu. Top down FP-growth for association rule mining. In *PAKDD '02: Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pages 334–340, London, UK, 2002. Springer-Verlag.
28. T. Yoshizawa, I. Pramudiono, and M. Kitsuregawa. SQL based association rule mining using commercial RDBMS (IBM DB2 UBD EEE). In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pages 301–306. Springer-Verlag, 2000.