Efficient Item Set Mining Supported by IMine Index

Manohar.M , Bharathi.G
Department of Computer Science and Engineering
Gudlavalleru Engineering College
Gudlavalleru—521356

***Abstract*-This paper presents the IMine index, a general and compact structure which provides tight integration of item set extraction in a relational DBMS. Since no constraint is enforced during the index creation phase, IMine provides a complete representation of the original database. To reduce the I/O cost, data accessed together during the same extraction phase are clustered on the same disk block. The IMine index structure can be efficiently exploited by different item set extraction algorithms. In particular, IMine data access methods currently support the FP-growth and LCM v.2 algorithms, but they can straightforwardly support the enforcement of various constraint categories. The IMine index has been integrated into the PostgreSQL DBMS and exploits its physical level access methods. Experiments, run for both sparse and dense data distributions, show the efficiency of the proposed index and its linear scalability also for large data sets. Item set mining supported by the IMine index shows performance always comparable with, and often (especially for low supports) better than, state-of-the-art algorithms accessing data on flat file.**

Index Terms — Data mining, item set extraction, indexing.

1 INTRODUCTION

ASSOCIATION rule mining discovers correlations among data items in a transactional database D. Each transaction in D is a set of data items. Association rules are usually represented in the form A→ B, where A and B are item sets, i.e., sets of data items. Item sets are characterized by their frequency of occurrence in D, which is called support. The data to be analyzed is usually stored into binary files, possibly extracted from a DBMS. Most algorithms [1], [2], [3], [4], [5] exploit ad hoc main memory data structures to efficiently extract item sets from a flat file. Recently, disk-based extraction algorithms have been proposed to support the extraction from large data sets [6], [7] but still dealing with data stored in flat files. To reduce the computational cost of item set extraction, different constraints may be enforced [8], [9], [10], [11], among which the most simple is the support constraint, which enforces a threshold on the minimum support of the extracted item sets.

Relational DBMSs exploit indices, which are ad hoc data structures, to enhance query performance and support the execution of complex queries. In this paper, we propose a similar approach to support data mining queries. The IMine index (Item set-Mine index) is a novel data structure that provides a compact and complete representation of transactional data supporting efficient item set extraction from a relational DBMS. It is characterized by the following properties:

1. It is a covering index. No constraint (e.g., support constraint) is enforced during the index creation phase.
2. The IMine index is a general structure which can be efficiently exploited by various item set extraction algorithms.
3. The IMine physical organization supports efficient data access during item set extraction.
4. IMine supports item set extraction in large data sets.

The IMine index has been implemented into the PostgreSQL open source DBMS [13]. Index data are accessed through PostgreSQL physical level access methods. The index performance has been evaluated by means of a wide range of experiments with data sets characterized by different size and data distribution. The execution time of frequent item set extraction based on IMine is always comparable with, and often (especially for low supports) faster than, the state-of-the-art algorithms (e.g., Prefix-Tree [14] and LCM v.2 [12]) accessing data on flat file.

This paper is organized as follows: Section 2 thoroughly describes the IMine index by addressing its structure, its data access methods, and its physical layout. Section 3 describes how the FP-growth and LCM v.2 algorithms may exploit IMine to perform efficiently the extraction of item sets. Section 4. Section 5 compares our approach with previous work. Finally, Section 6 draws conclusions and presents future developments of the proposed approach.

2 THE IMINE INDEX

The transactional data set D is represented, in the relational model, as a relation R. Each tuple in R is a pair (TransactionID, ItemID). The IMine index provides a compact and complete representation of R. Hence, it allows the efficient extraction of item sets from R, possibly enforcing support or other constraints.

2.1 IMine Index Structure

The structure of the IMine index is characterized by two components: the Item set-Tree and the Item-Btree. The two components provide two levels of indexing. In the following, we describe in more detail the I-Tree and the I-Btree structures.

TID	ItemsID
1	g,b,h,e,p,v,d
2	e,m,h,n,d,b
3	p,e,c,i,f,o,h
4	j,h,k,a,w,e
5	n,b,d,e,h

TID	ItemsID
6	s,a,n,r,b,u,i
7	b,g,h,d,e,p
8	a,i,b
9	f,i,e,p,c,h
10	t,h,a,e,b,r

TID	ItemsID
11	a,r,e,b,h
12	z,b,i,a,n,r
13	b,e,d,p,h

Fig. 1. Example data set.

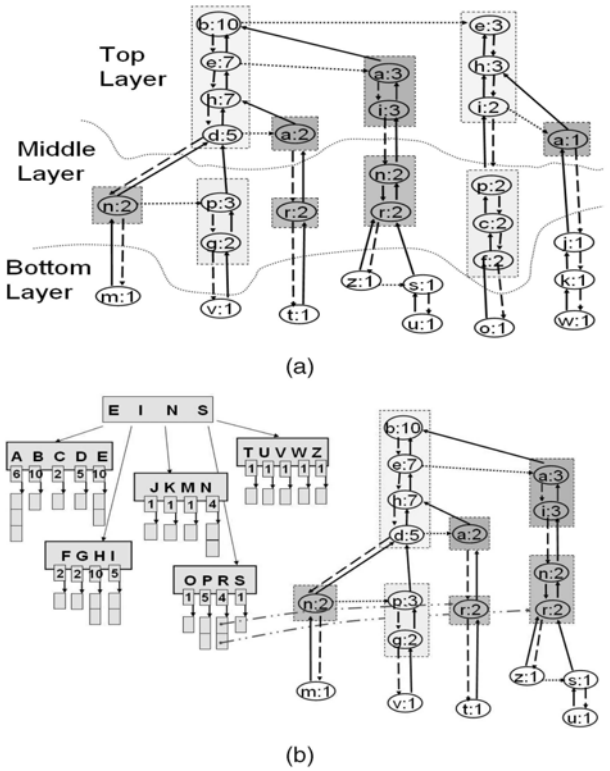


Fig. 2. IMine index for the example data set. (a) I-Tree. (b) I-Btree.

2.1.1 I-Tree

An effective way to compactly store transactional records is to use a prefix-tree. Our current implementation of the I-Tree is based on the FP-tree data structure [3], which is very effective in providing a compact and lossless representation of relation R. However, since the two index components are designed to be independent, alternative I-Tree data structures can be easily integrated in the IMine index.

The I-Tree associated to relation R is actually a forest of prefix-trees, where each tree represents a group of transactions all sharing one or more items. Each node in the I-Tree corresponds to an item in R. Each path in the I-Tree is an ordered sequence of nodes and represents one or more transactions in R. Each item in relation R is associated to one or more I-Tree nodes and each transaction in R is represented by a unique I-Tree path.

Fig.1 reports (in a more succinct form than its actual relational representation) a small data set used as a running example, and Fig.2 shows the complete structure of the corresponding IMine index. In the I-Tree paths (Fig. 2a), nodes are sorted by decreasing support of the corresponding items. In the case of items with the same support, nodes are sorted by item lexicographical order. In the I-Tree, the common prefix of two transactions is represented by a single path. For instance, consider transactions 3, 4, and 9 in the example data set. These transactions, once sorted as described above, share the common prefix [e: 3, h: 3], which is a single path in the I-Tree. Node [h: 3] is the root of two subpaths, representing the remaining items in the considered transactions. Each I-Tree node is associated with a node support value, representing the number of transactions which contain (without any different interleaved item) all the items in the subpath reaching the node. For example, in subpath [e: 3, h: 3], the support of node [h: 3] is 3.Hence, this subpath represents three transactions (i.e., transactions 3, 4, and 9). Each item is associated to one or more nodes. The item support is obtained by adding the support of all nodes including the item.

Nodes in the I-Tree are linked by means of pointers which allow selectively loading from disk the index portion necessary for the extraction task. Each node contains three pointers to nodes in the tree. Each pointer stores the physical location of the corresponding node. An arbitrary node (e.g., [p:3] in the example I-Tree in Fig. 2a) includes the following links: 1) Parent pointer (continuous edge linking node [p:3] to node [d:5]). 2) First child pointer (dashed edge linking node [p: 3] to node [g: 2]). When a node has more direct descendants, this pointer points to the first child node inserted in the I-Tree. 3) Right brother pointer (dotted edge linking node [p: 3] to node [n: 2]). When a node has many brothers (i.e., direct descendants of the same father), the pointer points to the first brother node inserted in the I-Tree after the current node. These pointers allow both bottom-up and top-down tree traversal, thus enabling item set extraction with various types of constraints (see Section 3).

The I-Tree is stored in the relational table $T_{\text{I-Tree,}}$ which contains one record for each I-Tree node. Each record contains node identifier, item identifier, node support, and pointers to the parent, first child, and right brother nodes. Each pointer stores the physical location (block number and offset within the block) of the record in table $T_{\text{I-Tree}}$ representing the corresponding node.

2.1.2 I-Btree

The I-Btree allows selectively accessing the I-Tree disk blocks during the extraction process. It is based on a B+Tree structure [15]. Fig. 2b shows the I-Btree for the example data set and a portion of the pointed I-Tree. For each item i in relation R, there is one entry in the I-Btree. In particular, the I-Btree leaf associated to i contains i's item support and pointers to all nodes in the I-Tree associated to item i. Each pointer stores the physical location of the record in table $T_{\text{I-Tree}}$ storing the node. Fig. 2b shows the pointers to the I-Tree nodes associated to item r.

2.2 IMine Data Access Methods

The IMine index structure is independent of the adopted item set extraction algorithm. Hence, different state-of-the-art algorithms may be employed, once data has been loaded in memory. The in-memory representation suitable for the selected extraction algorithm is employed (e.g., FP-tree for FP-growth, array-based structure for LCM). Depending on the enforced support and/or item constraints and on the selected algorithm for item set extraction, a different portion of the IMine index should be accessed. We devised three data access methods to load from the IMine index the following projections of the original database: 1) Frequent-item based projection, to support projection-based algorithms (e.g., FP-growth [3]). 2) Support-based projection, to support level based (e.g., APRIORI [1]), and array-based (e.g., LCM v.2 [12]) algorithms. 3) Item-based projection, to load all transactions where a specific item occurs, enabling constraint enforcement during the extraction process. The three access methods are described in the following sections.

2.2.1 Loading the Frequent-Item Projected Database

The Load_Freq_Item_Projected_DB access method reads the frequent-item projected database (see Fig. 3). It is based on the function Load_ Prefix _Path which loads a node

prefix path by a bottom-up I-Tree visit exploiting the node parent pointer. First, the I-Tree paths containing item α are identified. By means of the I-Btree, the pointers to all the I-Tree nodes for item α are accessed (function get_I-Btree_leaves in line 1) and the corresponding nodes are read from table $T_{I\text{-}Tree}$. Then, for each node, its prefix path is read (function Load Prefix Path in line 6). Starting from a given node, the visit goes up the I-Tree by following the node parent pointer until the tree root is reached (lines 10-12).Once read, prefix paths are stored in an in-memory representation of the projected database denoted as D_α(line 7). In each prefix path, node supports are normalized to the node support of item α in the subpath to avoid considering transactions not including α.

2.2.2 Loading the Support-Based Projected Database

The Load_Support_Projected_DB data access method loads the support-projected database (see Fig. 4). It is based on the function Load_SubTree, which reads a node subtree by means

```
Load_Freq_Item_Projected_DB(α,D_α,T_I-Tree,I-Btree) {
  /*read pointers to all nodes associated to α*/
1. α.NodePointers=get_I-Btree_leaves(α,I-Btree);
  /*read pointer to first index node associated to α*/
2. NodePointer=get_next_Pointer(α.NodePointers);
3. do {
    /*read node associated to α from table T_I-Tree */
4.   N=read_record(T_I-Tree,NodePointer);
    /*read prefix path from node N up to I-Tree root*/
5.   Path=N;
6.   Load_Prefix_Path(Path,N,T_I-Tree);
    /*add the path to the projected database D_α*/
7.   add_Path_to_D_α(Path,D_α);
    /*read pointer to next index node (if any) for α*/
8.   NodePointer=get_next_Pointer(α.NodePointers);
9. }while (NodePointer is not NULL);}

Load_Prefix_Path(Path,N,T_I-Tree) {
   /*read bottom-up the prefix path up to the I-Tree root*/
10. while (N.ParentPointer is not NULL) {
    /*read the parent node of N*/
11.   N=read_record(T_I-Tree,N.ParentPointer);
12.  add_Node_to_Path(N,Path); }}
```

Fig. 3. Loading the frequent-item projected database.

```
Load_Support_Projected_DB(MinSup,D_MinSup, T_I-Tree,I-Btree) {
  /*compute the frequent item list I_MinSup*/
1. I_MinSup = get_freq_items(MinSup, I-Btree);
  /*for each I-Tree root, load its subtree*/
2. N=get_first_I-Tree_root(T_I-Tree);
3. do {
4.   Path=∅;
5.   Load_SubTree(Path,N,I_MinSup,D_MinSup,T_I-Tree);
    /*read the next root node (if any), otherwise return
     NULL*/
6.   N=read_record(T_I-Tree,N.BrotherPointer);
7. } while (N is not NULL); }
Load_SubTree(Path,N,I_MinSup,D_MinSup,T_I-Tree){
  /*if the item is unfrequent, the path is stored in
   D_MinSup */
8. if (N.Item ∉ I_MinSup) then {
9.   add_Path_to_D_MinSup(Path,D_MinSup);
10.  return; }
  /*if the item is frequent, visit its subtree*/
11. add_Node_to_Path(N,Path);
  /*read the child node if any*/
12. if (N.ChildPointer is not NULL) then {
13.   N1=read_record(T_I-Tree,N.ChildPointer);
14.   Load_SubTree(Path,N1,I_MinSup,D_MinSup,T_I-Tree);}
15. else {
   /*if there are no child nodes, the path is stored in
    D_MinSup*/
16.   add_Path_to_D_MinSup(Path,D_MinSup); }
  /*read the brother node (if any)*/
17. if (N.BrotherPointer is not NULL) {
18.   N1=read_record(T_I-Tree,N.BrotherPointer);
19.   Load_SubTree(Path,N1,I_MinSup,D_MinSup,T_I-Tree);}}
```

Fig. 4. Loading the support-projected database.

of a top-down depth-first I-Tree visit exploiting both the node child and brother pointers. First frequent items are stored in set I_{MinSup}. Item support is read from the appropriate I-Btree leaf (function get_freq_items in line 1).Then, function Load_SubTree is invoked on each I-Tree root to read it's the Load_Support_Projected_DB data access method loads the support-projected database (see Fig. 4).It is based on the function Load_SubTree, which reads a node subtree by means of a top-down depth-first I-Tree visit exploiting both the node child and brother pointers. First frequent items are stored in set subtree (line 5). Starting from a root node, the I-Tree is visited depth-first by following the node child pointer (lines 12-14). The visit ends when a node with an unfrequent item (lines 8-10) or a node with no children (lines 15-16) is reached. The read subpath is added to the in memory representation of the projected database denoted as D_{MinSup} .Then, the search backtracks to the most recent node whose exploration is not finished, i.e., a node with (at least) one brother node. By following the brother pointer in the node, the brother node is read and the visit of the I-Tree is restarted from there (lines 17-19). Since the I-Tree roots are linked by brother pointers, when one root subtree has been completely explored, the next root is reached (line 6) and the Load_SubTree function is invoked on it.

2.2.3 Loading the Item-Projected Database

It exploits both the Load Prefix Path and Load SubTree functions previously described in Sections 2.2.1 and 2.2.2. The database transactions including a given item α are represented by the I-Tree paths containing α. These paths are selectively identified by means of the I-Btree, which returns the pointers to all nodes for α. For each node, first its prefix path is loaded by using the function Load_Prefix_Path. Then, its subtree is read by means of the function Load_SubTree. Each path obtained by the prefix path and one of the subpaths in the subtree represents a set of (complete) transactions including item α.

2.3 IMine Physical Organization

The physical organization of the IMine index is designed to minimize the cost of reading the data needed for the current extraction process. The I-Btree allows a selective access to the I-Tree paths of interest. Hence, the I/O cost is mainly given by the number of disk blocks read to load the required I-Tree paths.

To reduce the I/O cost, correlated index parts, i.e., parts that are accessed together during the extraction task should be clustered into the same disk block. The I-Tree physical organization is based on the following correlation types:

- *Intratransaction correlation.* Extraction algorithms consider together items occurring in the same transaction. Items appearing in a transaction are thus intrinsically correlated. To minimize the number of read blocks, each I-Tree path should be partitioned in a limited number of blocks.
- *Intertransaction correlation.* Transactions with some items in common will be accessed together when item sets including the common items are extracted.Hence; they should be stored in the same blocks.

2.3.1 I-Tree Layers

The I-Tree is partitioned in three layers based on the node access frequency during the extraction processes. The three layers are shown in Fig. 2a for the example I-Tree.

Top layer. This layer includes nodes that are very frequently accessed during the mining process. These nodes are located in the upper levels of the I-Tree. They correspond to items with high support, which are distributed over few nodes with high node support.

Middle layer. This layer includes nodes that are quite frequently accessed during the mining process. These nodes are typically located in the central part of the tree.

Bottom layer. This layer includes the nodes corresponding to rather low support items, which are rarely accessed during the mining process.

2.3.2 I-Tree Path Correlation

Correlation among the subpaths within each layer is analyzed to optimize the index storage on disk. Two paths are correlated when a given percentage of items is common to both paths.Each node may be shared by many paths; redundancy in storing the paths might be introduced. To prevent this effect, paths are partitioned in nonoverlapped parts, named tracks. Each node (even if shared among several paths) belongs to a single track. Correlation between track pairs is then analyzed.

Tracks are computed separately in each layer. Each layer is bound by two borders, named upper and lower border, which contain, respectively, the root and the tail nodes for the subpaths in the layer. For a given layer, track computation starts from nodes in its lower border. Each node in the (lower) border is the tail node of a different track. Nodes are considered based on their order into the lower border. For each tail node, its prefix-path is visited bottom-up. The visit ends when a node already included in a previously computed track or included in the upper border of the layer is reached. All visited nodes are assigned to the new track.

Correlation analysis is performed both in the Top and Middle layers, which contain paths associated to items with medium-high support. In the Bottom layer, instead, paths usually share a negligible number of items and correlation analysis becomes useless.

3 ITEM SET MINING

The IMine index is a disk resident data structure, the process is structured in two sequential steps: 1) the needed index data is loaded and 2) item set extraction takes place on loaded data. Once data are in memory, the appropriate algorithm for item set extraction can be applied.

3.1 Frequent Item Set Extraction

We present two approaches, denoted as FP-based and LCM-based algorithms, which are an adaptation of the FP-Growth algorithm [3] and LCM v.2 algorithm [12], respectively.

FP-based algorithm. The FP-growth algorithm [3] stores the data in a prefix-tree structure called FP-tree. First, it computes item support. Then, for each transaction, it stores in the FP-tree its subset including frequent items. Items are considered one by one. For each item, extraction takes place on the frequent-item projected database, which is generated from the original FP-tree and represented in a FP-tree based structure.

The FP-based algorithm selects frequent items by means of the get_freq_items function. For each item, the corresponding projected database is loaded from the IMine index by means of the Load_Freq_Item_Projected_DB access method (Section 2.2.1). Then, the original FP-growth algorithm [3] is run. With respect to [3], the FP-based approach reduces memory occupation by loading in memory only the projection exploited in the current extraction phase. Hence, more memory space is available for the extraction process.Data access overhead has been further reduced by exploiting correlation.

LCM-based algorithm.LCM v.2 algorithm [12]loads in memory the support-based projection of the original database. First, it reads the transactions to count item support. Then, for each transaction, it loads the subset including frequent items. Data are represented in memory by means of an array-based data structure, on which the extraction takes place.

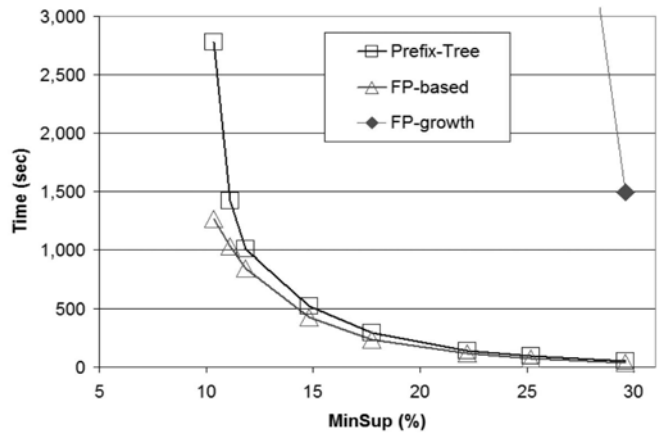
In the LCM-based algorithm, the database projection is read from the IMine index by means of the Load_Support_Projected_DB access method (Section 2.2.2). Data are stored in the appropriate array-based structure, on which the original LCM v.2 algorithm [12] is run. Since I-Tree paths concisely represent transactions, reading the database projection from the IMine index instead of from the original database is more effective in large databases.

4 EXPERIMENTAL RESULTS

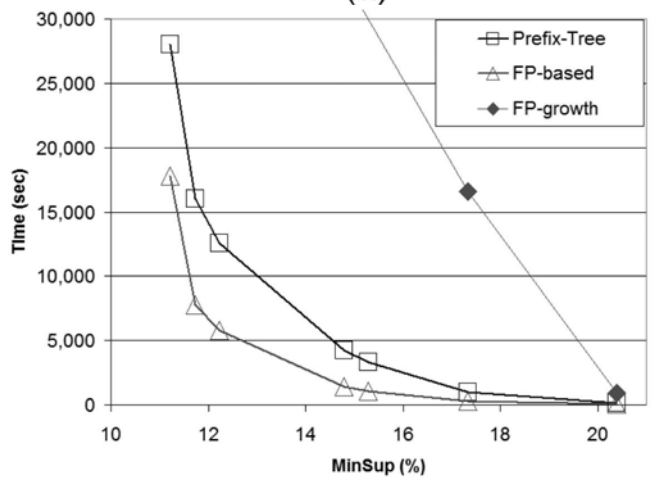
We ran the experiments for both dense and sparse data distributions. Connect and Pumsb [16] are dense and medium-size data sets. Kosarak [16] is a large and sparse data set including click-stream data. For all data sets, the index has been generated without enforcing any support threshold.Both the index creation procedure and the item set extraction algorithms are coded in Java language by using swings on Windows operating system.

Fig. 6 compares the FP-based algorithm with the Prefix-tree [14] and FP-growth algorithms [3] on flat file, all characterized by a similar extraction approach. For real data sets (Connect, Pumsb, and Kosarak), differences in CPU time between the FP-based and the Prefix-Tree algorithms are not visible for high supports, while for low supports the FP-based approach always performs better than Prefix-Tree. The FP-based algorithm, albeit implemented into a relational DBMS, yields performance always comparable with and sometimes better than the other algorithms.

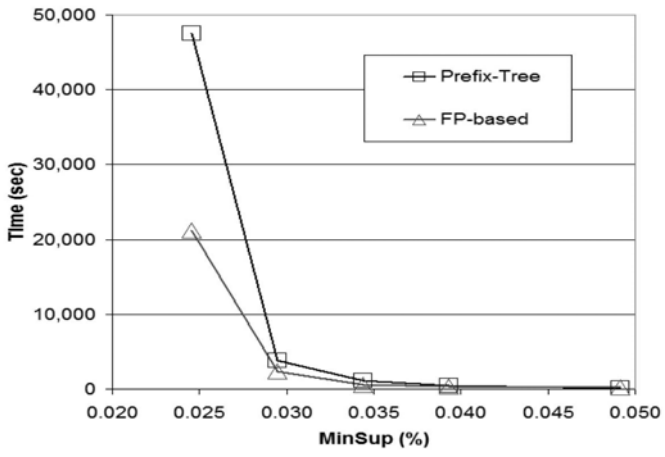
As shown in Fig. 7, the LCM-based approach provides an extraction time comparable to LCM v.2 on flat file. For large data sets, it performs better than LCM v.2. Since I-Tree paths compactly represent the transactions, reading the needed data through the index requires a lower number of I/O operations with respect to accessing the flat file representation of the data set. This benefit increases when the data set is larger and more correlated.



(a)

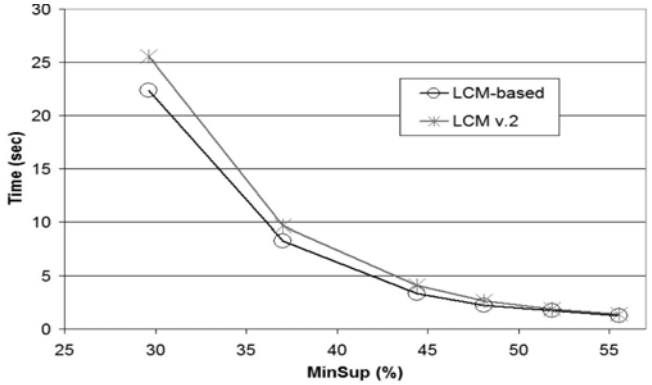


(b)

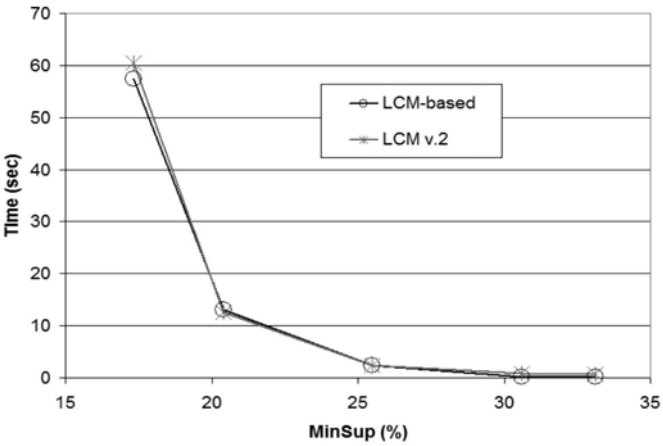


(c)

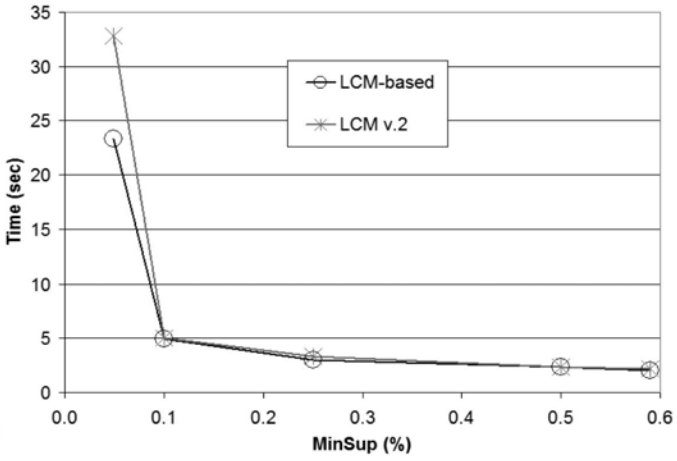
Fig. 6. Frequent item set extraction time for the FP-based algorithm. (a) Connect. (b) Pumsb. (c) Kosarak.



(a)



(b)



(c)

Fig. 7. Frequent item set extraction time for the LCM-based algorithm. (a) Connect. (b) Pumsb. (c) Kosarak.

4.1 Memory Consumption

On the Kosarak data set, the peak main memory is always significantly higher for the Prefix-Tree than for the FP-based algorithm, while on the Pumsb data set the difference between the two approaches is less relevant. The FP-based approach selectively loads in memory only the data required for the current extraction phase.

5 CONCLUSIONS AND FUTURE WORK

The IMine index is a novel index structure that supports efficient item set mining into a relational DBMS. It has been implemented into the PostgreSQL open source DBMS, by exploiting its physical level access methods. The IMine index provides a complete and compact representation of transactional data. It is a general structure that efficiently supports different algorithmic approaches to item set

extraction. Selective access of the physical index blocks significantly reduces the I/O costs and efficiently exploits DBMS buffer management strategies. This approach, albeit implemented into a relational DBMS, yields performance better than the state-of-the-art algorithms (i.e., Prefix-Tree [14] and LCM v.2 [12]) accessing data on a flat file and is characterized by a linear scalability also for large data sets.

As further extensions of this work, the following issues may be addressed: 1) Compact structures suitable for different data distributions. 2) Integration with a mining language. 3) Incremental update of the index.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast Algorithm for Mining Association Rules," Proc. 20th Int'l Conf. Very Large Data Bases (VLDB '94), Sept. 1994.

[2] R. Agrawal, T. Imilienski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," Proc. ACM SIGMOD '93, May 1993.

[3] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," Proc. ACM SIGMOD, 2000.

[4] H. Mannila, H. Toivonen, and A.I. Verkamo, "Efficient Algorithms for Discovering Association Rules," Proc. AAAI Workshop Knowledge Discovery in Databases (KDD '94), pp. 181-192, 1994.

[5] A. Savasere, E. Omiecinski, and S.B. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases,"Proc. 21st Int'l Conf. Very Large Data Bases (VLDB '95), pp. 432-444, 1995.

[6] G. Grahne and J. Zhu, "Mining Frequent Itemsets from Secondary Memory," Proc. IEEE Int'l Conf. Data Mining (ICDM '04), pp. 91-98, 2004.

[7] G. Ramesh, W. Maniatty, and M. Zaki, "Indexing and Data Access Methods for Database Mining," Proc. ACM SIGMOD Workshop Data Mining and Knowledge Discovery (DMKD), 2002.

[8] Y.L. Cheung, "Mining Frequent Itemsets without Support Threshold: With and without Item Constraints," IEEE Trans.Knowledge and Data Eng., vol. 16, no. 9, pp. 1052-1069, Sept. 2004.

[9] G. Cong and B. Liu, "Speed-Up Iterative Frequent Itemset Mining with Constraint Changes," Proc. IEEE Int'l Conf. Data Mining (ICDM '02), pp. 107-114, 2002.

[10] C.K.-S. Leung, L.V.S. Lakshmanan, and R.T. Ng, "Exploiting Succinct Constraints Using FP-Trees,"SIGKDD Explorations Newsletter,vol. 4, no. 1, pp. 40-49, 2002.

[11] R. Srikant, Q. Vu, and R. Agrawal, "Mining Association Rules with Item Constraints," Proc.Third Int'l Conf. Knowledge Discovery and Data Mining (KDD '97), pp. 67-73, 1997.

[12] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets,"Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations (FIMI), 2004.

[13]POSTGRESQL, http://www.postgresql.org, 2008.

[14] G. Grahne and J. Zhu, "Efficiently Using Prefix-Trees in Mining Frequent Itemsets," Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations (FIMI '03), Nov. 2003.

[15] R. Bayer and E.M. McCreight, "Organization and Maintenance of Large Ordered Indices," Acta Informatica, vol. 1, pp. 173-189, 1972.

[16] FIMI, http://fimi.cs.helsinki.fi/, 2008.