ISSN:0975-9646

Item Set Extraction of Mining Association Rule

Shabana Yasmeen, Prof. P.Pradeep Kumar, A.Ranjith Kumar
Department CSE, Vivekananda Institute of Technology and Science, Karimnagar, A.P, India

Abstract: Most of the research activities in association rule mining focuses on defining efficient algorithms for item set extraction. To reduce the computational complexity of item set extraction, support constraint is enforced on the extracted item sets.The IMine index structure can be efficiently exploited by different item set extraction algorithms. 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 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 lowsupports) better than, state-of-the-art algorithms accessing data on flat file.

Key Words: 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. Research activity usually focuses on defining efficient algorithms for item set extraction, which represents the most computationally intensive knowledge extraction task in association rule mining [1]. The data to be analyzed is usually stored into binary files, possibly extracted from a DBMS. The proposed work presents an incremental update strategy to work on the dynamic transaction of DMBS for efficient item set extraction. Since no support threshold is enforced during the index creation phase, the incremental update is feasible without accessing the original transactional database. The index performance in terms of incremental updates is experimentally evaluated with data sets characterized by different size and data distribution. The execution time of frequent item set extraction based on incremental update strategy of IMine is better than the state-of-the-art algorithm i.e., existing IMine algorithm without update strategy. The experimental result shows the scalability of incremental update strategy for more frequent database updates characterized by a large number of transactions and with different pattern lengths. Most algorithms [1], [2], [3], [4], [5], [6] 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 [7], [8], [9], but still dealing with data stored in flat files. To reduce the computational cost of item set extraction, different constraints may be enforced [10], [11], [12], [13], 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. Hence, the extraction can be performed by means of the index alone, without accessing the original database. The data representation is complete and allows reusing the index for mining item sets with any support threshold.
2. The IMine index is a general structure which can be efficiently exploited by various item set extraction algorithms. These algorithms can be characterized by different in-memory data representations (e.g., array list, prefix-tree) and techniques for visiting the search space. Data access functions have been devised for efficiently loading in memory the index data. Once in memory, data is available for item set extraction by means of the algorithm of choice. We implemented and experimentally evaluated the integration of the IMine index in FP-growth [3] and LCM v.2 [14]. Furthermore, the IMine index also supports the enforcement of various constraint categories [15].
3. The IMine physical organization supports efficient data access during item set extraction. Correlation analysis allows us to discover data accessed together during pattern extraction. To minimize the number of physical data blocks read during the mining process, correlated information is stored in the same block.
4. IMine supports item set extraction in large data sets. We exploit a direct writing technique to avoid representing in memory the entire large data set. Direct materialization has a limited impact on the final index size because it is applied only on a reduced portion of the data set (the less frequent part).

The IMine index has been implemented into the PostgreSQL open source DBMS [16]. 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 [17] and LCM v.2 [14]) accessing data on flat file. Furthermore, the experimental results show the linear scalability of both IMine-based algorithms also for data sets characterized by a large number of transactions and different pattern length.

TID	ItemsID	TID	ItemsID	TID	ItemsID
1	g,b,h,e,p,v,d	6	s,a,n,r,b,u,i	11	a,r,e,b,h
2	e,m,h,n,d,b	7	b,g,h,d,e,p	12	z,b,i,a,n,r
3	p,e,c,i,f,o,h	8	a,i,b	13	b,e,d,p,h
4	j,h,k,a,w,e	9	f,i,e,p,c,h		
5	n,b,d,e,h	10	t,h,a,e,b,r		

Fig. 1. Example data set.

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. It also describes how the IMine index supports the enforcement of various constraint types.

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. In Section 2.1, we present the general structure of the IMine index; while in Section 2.2, we discuss how data access takes place. The physical organization of the index is presented in Section 2.3 together with a discussion of access cost. Finally, Section 2.4 discusses some optimizations for the physical storage of large sparse data sets.

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. The Item set-Tree (I-Tree) is a prefix-tree which represents relation R by means of a succinct and lossless compact structure. The Item-Btree (I-Btree) is a B+Tree structure which allows reading selected I-Tree portions during the extraction task. For each item, it stores the physical locations of all item occurrences in the I-Tree. Thus, it supports efficiently loading from the I-Tree the transactions in R including the item. In the following, we describe in more detail the I-Tree and the I-Btree structures. I-Tree An effective way to compactly store transactional records is to use a prefix-tree. Trees and prefix-trees have been frequently used in data mining and data warehousing indices, including cube forest [18], FP-tree [3], H-tree [19], Inverted Matrix [7], and Patricia-Tries [20]. 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)

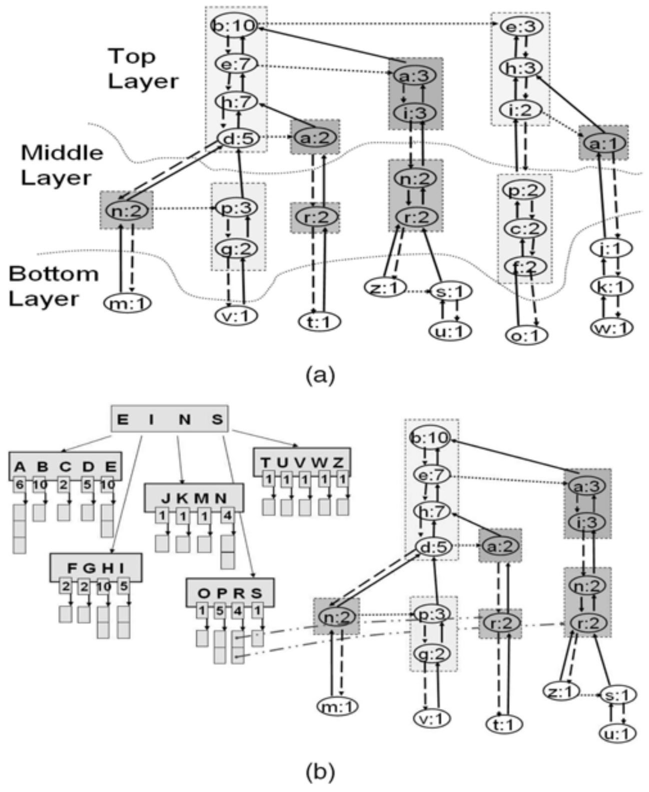


Fig. 2. IMine index for the example data set. (a) I-Tree. (b) I-Btree.

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]). 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 TI_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 TI_Tree representing the corresponding node.

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 [21]. 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 TI_Tree storing the node. Fig. 2b shows the pointers to the I-Tree nodes associated to item r.

2.2 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.

When visiting the I-Tree, nodes are read from table TI_Tree by using their exact physical location. However, fetching a given record requires loading the entire disk block where the record is stored. On the other hand, once the block is in the DBMS buffer cache, reading the other nodes in the block does not entail additional I/O cost. Hence, 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:

2.2.1 I-Tree Layers

The I-Tree is partitioned in three layers based on the node access frequency during the extraction processes. The frequency in accessing a node (and thus the subpath including it) depends on the interaction of three factors: 1) the node level in the I-Tree, i.e., its distance from the root,

2) the number of paths including it, represented by the node support, and 3) the support of its item. When an item has very low support, it will be very rarely accessed, because it will be uninteresting for most support thresholds. Nodes located in lower levels of the I-Tree are associated to items with low support. 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. Items are chosen in the same order they are entered in the I-Tree paths. The nodes containing the selected items are all stored in the Top layer.

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. They correspond to items with relatively high support, but not yet dispersed on a large number of nodes with very low node support. We include in the Middle layer nodes with (node) support larger than 1. Unitary support nodes are rather rarely accessed and should be excluded from the Middle layer.

Bottom layer. This layer includes the nodes corresponding to rather low support items, which are rarely accessed during the mining process. Nodes in this layer are analyzed only when mining frequent item sets for very low support thresholds. The Bottom layer is characterized by a huge number of paths which are (possibly long) chains of nodes with unitary support. These subpaths represent (a portion of) a single transaction and are thus read only few times. A large number of low support items is included in this layer.

2.2.2 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. Searching for optimal correlation is computationally expensive since all pairs of paths should be checked. As an alternative, we propose a heuristic technique to detect correlation with reduced computation cost. The technique is based on an "asymmetric" definition of correlation. A reference path, named pivot, is selected. Then, correlation of the other paths with the pivot is analyzed. The pivot and its correlated paths are stored in the same disk block.

Since 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. The pseudocode for track computation is provided in Appendix A, which can be found on the Computer Society Digital

Library at http://doi.ieee computersociety .org/10.1109/ TKDE. 2008. 180.

As an example, Fig. 2a shows how paths in the Top and Middle layers are partitioned in tracks (tracks are represented as dashed boxes). The top layer upper border
contains nodes [b:10], [a:3], [e:3] (which correspond to the I-Tree roots) and the lower border includes nodes [d:5], [a:2], [i:3], [i:2], [a:1]. Track computation starts by considering node [d:5]. Its prefix-path [d:5, h:7, e:7, b:10] is a track. After considering node [a:2], the prefix path of node [i:3] is visited until node [b:10] is reached. Since [b:10] belongs to the previously computed track, the new track will only include subpath [i:3, a:3].

After each layer is partitioned in tracks, correlation analysis between track pairs may take place. The longest track that can be completely stored in the block is selected as pivot. Then, correlation between the remaining tracks and the pivot is computed. Only tracks that can completely fit in the remaining space in the block are considered, in decreasing length (i.e., number of nodes in the track) order. Tracks correlated to the pivot are stored in the same disk block. When no more tracks can be stored in the block or no remaining track is correlated with the current pivot, a new block and a new pivot are selected.

3. ITEM SET MINING

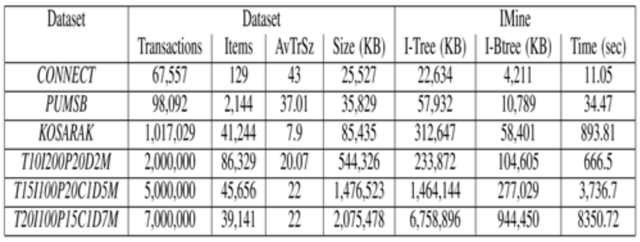
Several algorithms have been proposed for item set extraction. These algorithms are different mainly in the adopted main memory data structures and in the strategy to visit the search space. The IMine index can support all these different extraction strategies. Since the IMine index is a disk resident 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. The data access methods presented in Section 2.2 allow effectively loading the data needed for the current extraction phase. Once data are in memory, the appropriate algorithm for item set extraction can be applied. In Section 3.1, frequent item set extraction by means of two representative state-ofthe- art approaches, i.e., FP-growth [3] and LCM v.2 [14], is described. Section 3.2 discusses how the IMine index supports constraint enforcement.

Enforcing Constraints

Constraint specification allows the (human) analyst to better focus on interesting item sets for the considered analysis task. A significant research effort [10], [11], [12], [15], [13] has been devoted to the exploration of techniques to push constraint enforcement into the extraction process, thus allowing an early pruning of the search space and a more efficient extraction process. Constraints have been classified as antimonotonic, monotonic, succinct, and convertible [15]. These latter constraints (e.g., avg, sum) are neither antimonotonic nor monotonic, but they can be converted into monotonic or antimonotonic by an appropriate item ordering.

Constraint enforcement into the FP-growth algorithm is discussed in [15]. This approach can be straightforwardly supported by the IMine index. More specifically, the items of interest are selected by accessing the I-Btree. Once data are in memory, the _-projected database is built by including all items which follow _ in the item ordering required by constraint enforcement. Different constraint classes may require different item orderings, which enable early pruning for the considered constraint class. For example, for convertible constraints, the ordering exploited to convert the constraint is enforced. The appropriate extraction algorithm performs the extraction by recursive projections of the _-projected database. Constraints are enforced during the iteration steps.

TABLE 1
Data Set Characteristics and Corresponding Indices



Dataset	Dataset				IMine		
	Transactions	Items	AvTrSz	Size (KB)	I-Tree (KB)	I-Btree (KB)	Time (sec)
CONNECT	67,557	129	43	25,527	22,634	4,211	11.05
PUMSB	98,092	2,144	37.01	35,829	57,932	10,789	34.47
KOSARAK	1,017,029	41,244	7.9	85,435	312,647	58,401	893.81
T10I200P20D2M	2,000,000	86,329	20.07	544,326	233,872	104,605	666.5
T15I100P20C1D5M	5,000,000	45,656	22	1,476,523	1,464,144	277,029	3,736.7
T20I100P15C1D7M	7,000,000	39,141	22	2,075,478	6,758,896	944,450	8350.72

4. EXPERIMENTAL RESULT

We validated our approach by means of a large set of experiments addressing the following issues:

1. Performance of the IMine index creation, in terms of both creation time and index size,
2. Performance of frequent item set extraction, in terms of Execution time, memory usage, and I/O access time, 2
3. Effect of the DBMS buffer cache size on hit rate, 3
4. Effect of the index layered organization,
5. Effect of direct writing, and
6. Scalability of the approach.

We ran the experiments for both dense and sparse data distributions. We report experiments on six representative data sets whose characteristics (i.e., transaction and item cardinality, average transaction size (AvgTrSz), and data set size) are in Table 1. Connect and Pumsb [22] are dense and medium-size data sets. Kosarak [22] is a large and sparse data set including click-stream data. T10I200P20D2M is a dense and large synthetic data set, while T15I100P20C1D5M and T20I100P15C1D7M are quite sparse and large synthetic data sets. Synthetic data sets are generated by means of the IBM generator [23]. 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 into the PostgreSQL v. 7.3.4 open source DBMS [16]. They have been developed in ANSI C. Experiments have been performed on a 2,800-MHz Pentium IV PC with 2-Gbyte main memory running Linux kernel v. 2.7.81. The buffer cache of the PostgreSQL DBMS has been set to the default size of 64 blocks (block size is 8 Kbytes). All reported execution times are real times, including both system and user time, and obtained from the Unix time command as in [22].

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 [17] and LCM v.2 [14]) 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. Currently, we adopt the prefix-tree structure to represent any transactional database independently of its data distribution. Different techniques may be adopted (e.g., [7]), possibly ad hoc for the local density of the considered data set portion. 2) Integration with a mining language. The proposed primitives may be integrated with a query language for specifying mining requests, thus contributing an efficient database implementation of the basic extraction statements. 3) Incremental update of the index. Currently, when the transactional database is updated, the IMine index needs to be rematerialized. A different approach would be to incrementally update the index when new data become available. Since no support threshold is enforced during the index creation phase, the incremental update is feasible without accessing the original transactional database.

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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] R. Srikant, Q. Vu, and R. Agrawal, “Mining Association Rule with Item Constraints,” Proc. Third Int’l Conf. Knowledge Discovery and Data Mining (KDD ’97), pp. 67-73, 1997.

[10] N. Agrawal, T. Imielinski, and A. Swami, “Database Mining: A Performance Perspective,” IEEE Trans. Knowledge and Data Eng., vol. 5, no. 6, Dec. 1993.

[11] S. Sarawagi, S. Thomas, and R. Agrawal, “Integrating Mining with Relational Database Systems: Alternatives and Implications,” Proc. ACM SIGMOD, 1998.

[12] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane, “DMQL: A Data Mining Query Language for Relational Databases,” Proc. ACM SIGMOD Workshop Data Mining and Knowledge Discovery (DMKD), 1996.

[13] M. Botta, J.-F. Boulicaut, C. Masson, and R. Meo, “A Comparison between Query Languages for the Extraction of Association Rules,” Proc. Fourth Int’l Conf. Data Warehousing and Knowledge Discovery (DaWak), 2002.