International Journal of Computer Science and Informatics

Volume 1 | Issue 1 Article 3

July 2011

Item set extraction without using constraints

K. Neelima
KL University Vaddeswaram,Guntur(Dt),A.P,India, neelima.kolli6@gmail.com

Y.Naga lakshmi
KL University Vaddeswaram,Guntur(Dt),A.P,India, lakshmi.yellela@gmail.com

Mr.M. Suman
KL University Vaddeswaram,Guntur(Dt),A.P,India, suman.maloji@gmail.com

Item set extraction without using constraints

K.Neelima, Y.Naga lakshmi, Mr.M.Suman
M.tech,Depatment of CSE
KL University
Vaddeswaram,Guntur(Dt),A.P,India.
neelima.kolli6@gmail.com, lakshmi.yellela@gmail.com, suman.maloji@gmail.com

***Abstract* - This paper presents the Item set extraction without using constraints, 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, It 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 index structure can be efficiently exploited by different item set extraction algorithms. In particular, data access methods currently support the FP-growth and LCM v.2 algorithms, but they can straightforwardly support the enforcement of various constraint categories. 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.**

Keywords-Data mining, item set extraction, indexing

I. 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. Most algorithms [1], [2], [3], [4]. 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 but still dealing with data stored in flat files. To reduce the computational cost of item set extraction, different constraints may be enforced among which the most simple is the support constraint, which enforces a threshold on the minimum support of the extracted item sets.

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.
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

II. I-TREE AND B-TREE STRUCTURE

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.

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

A. 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, Inverted Matrix , and Patricia-Tries . 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.

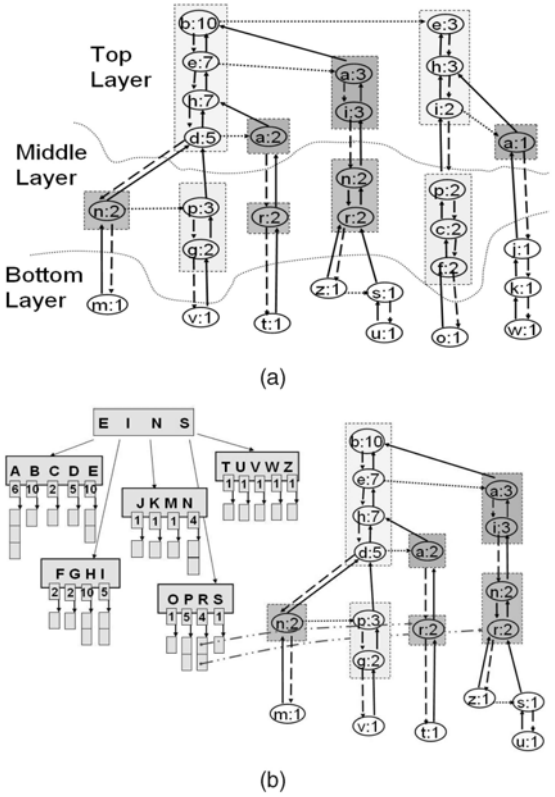

Fig. 2. IMine index for the example data set. (a) I-Tree. (b) I-Btree.

Fig. 1 reports 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 sub paths, representing the remaining items in the considered transactions.

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.

B. 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. 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_{I_Tree} storing the node. Fig. 2b shows the pointers to the I-Tree nodes associated to item r.

III. 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. 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 .

2) Support-based projection, to support level based

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.

```
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

Since IMine is a covering index, the original database is never accessed. The IMine index allows selectively loading into memory only the index blocks used for the local search. Hence, it supports a reduction of disk reads. Since only a small fragment of the data is actually loaded in memory, more memory space is available for the extraction task.

A. Loading the Frequent-Item Projected Database

The frequent-item projection of relation R with respect to an arbitrary item α includes the transactions in R where α occurs, intersected with the items having higher support than α or equal support but preceding α in lexicographical order [3]. In the I-Tree paths, items are sorted by descending support and lexicographical order. Thus, the projection is represented by the I-Tree prefix paths of item α (i.e., the subpaths from the I-Tree roots to the nodes with α). 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-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 _ in the subpath to avoid considering transactions not including α. In the example data set, item p appears in two nodes, i.e., [p:3] and [p:2]. The access method reads two prefix paths for p, i.e., [p : 3 → d : 5 →h : 7 → e : 7 → b : 10] and [p : 2 → i : 2 → h : 3 → e : 3] Each subpath is normalized to p node support. For example, the first prefix path, once normalized to [p:3], is [p : 3 →d : 3 →h : 3 → e : 3 → b : 3].

B. Loading the Support-Based Projected Database

The support-based projection of relation R contains all transactions in R intersected with the items which are frequent with respect to a given support threshold (MinSup). The I-Tree paths completely represent the database transactions. Items are sorted by decreasing support along the paths. Thus, the support-based projection of R is given by the I-Tree subpaths between the I-Tree roots and the first node with an

Un frequent item. 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 IMinSup. 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 its 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 DMinSup. 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.

C. Loading the Item-Projected Database

To support constraint enforcement (see Section 3.2), all the transactions including a given item should be selectively loaded. These transactions represent the item-projected database, which is read by the Load_Item_Projected_DB access method. 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. In the running example, consider item a with nodes [a:3], [a:1], and [a:2]. The prefix path of [a:3] is [a : 3 → b : 10] and its subtree includes the subpaths [a : 3 → i : 3 → n : 2 →r : 2 →z : 1] and [a : 3 → i : 3 → n : 2 → r : 2 → s : 1 → u : 1]. The paths representing the complete transaction sets including item a are [b:10, a:3, i:3, n:2, r:2, z:1] and [b:10, a:3, i:3, n:2, r:2, s:1, u:1].

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

IV. CONCLUSIONS AND FUTURE WORK

The IMine index is a novel index structure that supports efficient item set mining into a relational DBMS. 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.

V. 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.