IMine: Index Support for Item Set Mining

Elena Baralis, Tania Cerquitelli, and Silvia Chiusano

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 \( A \rightarrow B \), 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], [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.

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. The experiments, which evaluate the effectiveness of the proposed index, are presented in 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. 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.

2.1.1 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 $[e:3,h:3]$, 

![Fig. 1. Example data set.](image1.jpg)

![Fig. 2. IMine index for the example data set. (a) I-Tree. (b) I-Btree.](image2.jpg)
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 $T_{I-Tree}$, 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 $T_{I-Tree}$ 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 [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 (block number and offset within the block) of the record in table $T_{I-Tree}$ storing the node. Fig. 2b shows the pointers to the I-Tree nodes associated to item $i$.

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 [14]) 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.

---

**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. Read disk blocks are stored in the buffer cache memory of PostgreSQL. Table $T_{I-Tree}$ and the I-Btree are accessed by using the read functions available in the PostgreSQL access methods [16].

2.2.1 Loading the Frequent-Item Projected Database

The frequent-item projection of relation $R$ with respect to an arbitrary item $\alpha$ includes the transactions in $R$ where $\alpha$ occurs, intersected with the items having higher support than $\alpha$ or equal support but preceding $\alpha$ 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 $\alpha$ (i.e., the subpaths from the I-Tree roots to the nodes with $\alpha$).

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 $\alpha$ are identified. By means of the I-Btree, the pointers to all the I-Tree nodes for item $\alpha$ 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 $D_\alpha$ (line 7). In each prefix path, node supports are normalized to their support of all nodes including the item. The $Load_{Freq~Item~Projected_DB}$ access method reads two prefix paths for $\alpha$, i.e., $[p:3 \rightarrow d:5 \rightarrow h:7 \rightarrow e:7 \rightarrow b:10]$ and $[p:2 \rightarrow i:2 \rightarrow h:3 \rightarrow e:3]$. The $Parent$ pointer is always NULL until the tree root is reached.
2.2.2 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 \( (\text{MinSup}) \). 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 unfrequent item.

The \texttt{Load\_Support\_Projected\_DB} data access method loads the support-projected database (see Fig. 4). It is based on the function \texttt{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 \( \mathcal{I}_{\text{MinSup}} \). Item support is read from the appropriate I-Btree leaf (function \texttt{get\_freq\_items} in line 1). Then, function \texttt{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 in the 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, they should be stored in the same blocks. In the running example, consider item \( a \) with nodes \([a:3],[a:1]\), and \([a:2]\). The prefix path of \([a:3]\) is \([a:3 \rightarrow b:10]\) and its subtree includes the subpaths \([a:3 \rightarrow i:3 \rightarrow n:2 \rightarrow r:2 \rightarrow z:1]\) and \([a:3 \rightarrow i:3 \rightarrow n:2 \rightarrow r:2 \rightarrow s:1 \rightarrow u:1]\).

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.

When visiting the I-Tree, nodes are read from table \( T_{I-Tree} \) 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:

- **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 items sets including the common items are extracted. Hence, they should be stored in the same blocks. In the I-Tree, the common prefix of different transactions is represented by a single path. To further increase transaction clustering, subpaths with a given percentage of items in common should be stored in the same disk block.
We devised a physical organization for the I-Tree which exploits both correlation types. For intratransaction correlation, I-Tree paths are partitioned into three layers, based on node access frequency (Section 2.3.1). Nodes in each layer correspond to items within the same support range. Path correlation is then analyzed among the subpaths in each layer (Section 2.3.2). A general discussion of the I/O performed by IMine is presented in Section 2.3.3.

### 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 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. These items can be characterized by considering the average support of their nodes. As a first attempt, items whose average node support is larger than or equal to a given threshold $K_{avg}$ may be assigned to the Top layer. Unfortunately, average node support is not monotone with respect to the item ordering on I-Tree paths, i.e., by descending item support and ascending lexicographical order. For example, items $\{b, e, h\}$ and $\{i\}$, which, respectively, precede and follow item $a$ on the I-Tree paths, have all an average node support higher than $a$. Hence, by directly selecting items based on average node support, tree paths could be unintentionally broken by omitting items that do not meet the $K_{avg}$ threshold. To prevent this effect, items are selected according to the item ordering on the I-Tree paths. First, the number of items (#Items) with average node support $\geq K_{avg}$ is computed. Next, the first $\#Items$ most frequent items occurring in the I-Tree are selected. 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. The pseudocode for Top layer item selection is provided in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/Tkde.2008.180. $K_{avg}$ ranges between 1 and the total number of transactions. For $K_{avg} = 1$, the Top layer includes all items. Experiments run on different data sets (see Section 4) allowed us to select $K_{avg} = 1.2$. The first node in each path whose item is not assigned to the Top layer is the root of the paths in the Middle part of the tree.

In the example in Fig. 2a, let $K_{avg}$ be 2.5. Six items out of 22 have average node support $\geq K_{avg}$, i.e., $\{b, e, h, d, i, p\}$. Hence, the first six items occurring in the I-Tree paths are selected based on the item ordering, i.e., $\{b, e, h, d, i, p\}$. Nodes associated to these items are in the top layer. We observe that item $a$ has average node support equal to 2, and items $\{b, e, h\}$ and $\{i\}$, preceding and following $a$ on the I-Tree paths, have average node support $\geq 2.5$. If item selection were based on $K_{avg}$, item $a$ would be omitted and paths including it would be broken (e.g., subpath $[b:10, a:3, i:3]$). Finally, items $p$, $d$, and $i$ have the same support. However, only items $d$ and $i$ are selected since they precede $p$ in lexicographical order.

**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.

### 2.3.2 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. 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.ieeecomputersociety.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.

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. In the Top layer, a track is correlated to the pivot when at least 50 percent of its items belongs also to the pivot, while in the Middle layer, when at least 20 percent of its items belongs to the pivot. In the Bottom layer, a vertical (i.e., depth-first) storage of the paths is performed. Thus, the path structure is preserved, but without any further storage optimization. Experiments in Section 4 show that performing correlation analysis in this layer does not yield any benefit.

2.3.3 I/O Analysis for Index Data Access
I-Tree paths are partitioned in subpaths stored in a specific disk block. Loading a subpath costs a single disk access. Both the Load Prefix Path and Load SubTree functions (see Section 2.2) sequentially access the nodes in a subpath (either top-down or bottom up) without interleaving any other operation. The disk block containing the subpath is thus loaded when the first (or last) node in the path is read. Likely, it will be in the buffer cache when the other nodes are accessed.

Both layer partitioning and path correlation analysis address clustering subpaths potentially accessed together in the same block. Reading the other subpaths does not cause an additional cost if the block is still in the buffer cache. Both the Load Freq Item Projected DB and Load Support Projected_DB access methods load the index subpaths including items above a given support threshold. Since each index layer includes items within the same support range, data accessed together are compactly stored in the same block. In particular, in the case of high supports, only the Top layer is accessed. For medium support values, both the Top and Middle layers are read, while for low or no support thresholds, all three layers are read. Path correlation supports both methods, but is expected to particularly enhance the Load Freq Item Projected DB method, which reads all prefix-paths for a given item. Since path correlation groups, within each layer, subpaths having items in common, the nodes including a given item are distributed in few blocks.

Blocks storing the Top layer are frequently accessed. Hence, they have a high probability to remain in the buffer cache. In particular, the Load Freq Item Projected_DB method accesses each node whenever the prefix path for one of the nodes following it is loaded. Hence, nodes in the upper parts are more frequently accessed. The Load Support Projected_DB method frequently accesses the Top layer to read the roots and their (immediate) subpaths.

Consider the running example. Fig. 5 shows the physical organization of the blocks including the frequent-item projected database for item p. Item p has two prefix paths, both partitioned in two disk blocks, i.e., blocks #3 and #1. Thus, for both prefix paths, data access costs two disk reads. First, through the I-Btree, block #3 is loaded in the buffer cache to access node [p:3] and is read from the block. Then, by following the node parent pointer in [p:3], block #1 is loaded and subpath [d:5 → h:7 → e:7 → b:10] is read. Now, prefix path [p:3 → d:5 → h:7 → e:7 → b:10] is in memory. The next prefix path is then read. If the two blocks are still in the buffer cache, reading this prefix path does not require additional disk reads. Node [p:2] is read from block #3 and then subpath [i:2 → h:3 → e:3] is read from block #1. Prefix path [p:2 → i:2 → h:3 → e:3] is also in memory.

2.4 IMine Index Materialization for Large Databases
The IMine index provides a complete representation of the database, which supports the extraction of item sets satisfying various types of constraints. During the index creation process, the I-Tree is first completely built in main memory and next written on disk. In the case of large databases, the I-Tree might not completely fit in memory. Hence, we devised a technique to avoid storing in memory the complete I-Tree, by directly writing on disk a portion of it.

Large data sets are usually characterized by a huge number of transactions and a rather sparse data distribution. Their corresponding I-Tree representation is characterized by an increasing number of long subpaths whose nodes have support equal to 1. I-Tree subpaths with unitary node support are in the Bottom layer. They represent (part of) a single transaction and are therefore traversed only once during index creation. Correlation is not analyzed on these subpaths. Thus, each subpath, once generated, can be directly written on disk. Unfortunately, node support is not known until the I-Tree is completely built in memory. We propose a heuristic approach to identify these subpaths immediately after they are generated. It is based on item
support, which is already known when the I-Tree creation begins. A reference item support threshold $S_w$ is identified. All items with support lower than $S_w$ should appear only in I-Tree nodes with unitary node support. The subpaths starting with one of these items are directly written on disk.

Threshold $S_w$ is computed by means of the average value $\mu_{sup}$ of item support. We have observed experimentally that most items with support lower than $\mu_{sup}$ occur only in nodes with unitary node support. $\mu_{sup}$ is usually rather low (e.g., 0.02 percent) and the standard deviation is very high (i.e., at least three order of magnitude larger than $\mu_{sup}$). Threshold $S_w$ is computed as $S_w = K_{sup} \cdot \mu_{sup}$ where $0 \leq K_{sup} \leq 1$. When $K_{sup} = 1$, then $S_w = \mu_{sup}$, while when $K_{sup} = 0$, no I-Tree part is written on disk before the I-Tree creation ends. We experimentally set $K_{sup} = 0.05$ (see Section 4.5) to deal with data sets whose I-Tree does not fit in main memory.

This approach allows us to build the complete IMine index for large databases without enforcing any constraint, while also reducing index creation time. A slight overhead in terms of required disk space may be introduced when the data distribution is dense. In this case, some nodes may actually have support higher than one. Thus, a single index subpath is instead represented by two different paths. The I-Tree compactness may be reduced and some overhead in data access may be introduced. Experiments showed that this slight overhead does not affect data loading performance during the extraction phase.

### 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-of-the-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.

#### 3.1 Frequent Item Set Extraction

This section describes how frequent item set extraction takes place on the IMine index. 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 [14], 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. Items considered in sequence may occur in correlated index paths. When the next item is considered, its paths are already available in memory and do not have to be read again.

**LCM-based algorithm.** The LCM v.2 algorithm [14] 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 [14] 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.

#### 3.2 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. For each item $\alpha$, the transactions including it are read by means of the `Load_Item_Projected_DB` access method. Once data are in memory, the $\alpha$-projected database is built by including all items which follow $\alpha$ 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. Since the `Load_Item_Projected_DB` access method loads in memory the entire transaction set including $\alpha$, the required order can always be enforced. The appropriate extraction algorithm performs the extraction by recursive projections of the $\alpha$-projected database. Constraints are enforced during the iteration steps.
The approach in [15] can be easily extended to deal with multiple constraints. Constraints may belong to the same category (e.g., all antimonotonic) or to different categories (e.g., antimonotonic and monotonic). The IMine index provides a flexible information storage structure which can easily support also this extension.

4 Experimental Results

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.
3. effect of the DBMS buffer cache size on hit rate.
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 has a lower size, for T15I100P20C1D5M is quite equivalent, and for Kosarak and T20I100P15C1D7M is larger. The I-Btree contains pointers to all nodes in the I-Tree. Hence, its size is proportional to the total number of I-Tree nodes.

The experiments in Section 4.2 show that the index size does not negatively affect the extraction performance, because IMine includes additional information with respect to the original database (e.g., node pointers) which supports effective data access. Furthermore, since the index is stored on disk, only the data needed for the current extraction process are actually loaded in memory.

Table 1 also shows the index creation time, which is mainly due to path correlation analysis and storage of the index paths on disk. The first factor depends on the number of I-Tree paths. The latter, which is dominant, is proportional to the number of I-Tree nodes. Both contributions increase for larger and sparser data sets, for which the I-Tree structure is more compact. In sparse data sets (e.g., Kosarak), where data are highly correlated, the I-Tree structure is more complex. In sparse data sets (e.g., Kosarak), where data are weakly correlated, data compression is low and storing the I-Tree requires more disk blocks. For example, with respect to the original database, the I-Tree for T10I200P20D2M has a lower size, for T15I100P20C1D5M is quite equivalent, and for Kosarak and T20I100P15C1D7M is larger. The I-Btree contains pointers to all nodes in the I-Tree. Hence, its size is proportional to the total number of I-Tree nodes.

4.1 Index Creation and Structure

Table 1 reports both I-Tree and I-Btree size for the six data sets. The overall IMine index size is obtained by summing both contributions. The IMine indices have been created with the default value $K_{avg} = 1.2$. Furthermore, the Connect, Pumsb, Kosarak, and T10I200P20D2M data sets have been created with $K_{sup} = 0$, while large synthetic data sets with $K_{sup} = 0.05$. These parameters are further discussed in Sections 4.4 and 4.5.

The adopted I-Tree representation is more suitable for dense data distributions, for which it provides good data compression. In dense data sets (e.g., T10I200P20D2M) where data are highly correlated, the I-Tree structure is more compact. In sparse data sets (e.g., Kosarak), where data are weakly correlated, data compression is low and storing the I-Tree requires more disk blocks. For example, with respect to the original database, the I-Tree for T10I200P20D2M has a lower size, for T15I100P20C1D5M is quite equivalent, and for Kosarak and T20I100P15C1D7M is larger. The I-Btree contains pointers to all nodes in the I-Tree. Hence, its size is proportional to the total number of I-Tree nodes.

The experiments in Section 4.2 show that the index size does not negatively affect the extraction performance, because IMine includes additional information with respect to the original database (e.g., node pointers) which supports effective data access. Furthermore, since the index is stored on disk, only the data needed for the current extraction process are actually loaded in memory.

Table 1 also shows the index creation time, which is mainly due to path correlation analysis and storage of the index paths on disk. The first factor depends on the number of I-Tree paths. The latter, which is dominant, is proportional to the number of I-Tree nodes. Both contributions increase for larger and sparser data sets, for which the I-Tree size increases (see Section 4.4).

4.2 Frequent Item Set Extraction Performance

The IMine structure is independent of the extraction algorithm. To validate its generality, we compared the FP-based and LCM-based algorithms with three very effective state-of-the-art algorithms accessing data on flat file, i.e., Prefix-tree [17], LCM v.2 [14] (FIMI ’03 and FIMI ’04 best implementation algorithms), and FP-growth [3]. We analyzed the runtime with various supports. Since our objective is to

---

**TABLE 1**

Data Set Characteristics and Corresponding Indices

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

---

2. Because of space constraints, I/O access time is analyzed in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.180.

3. Because of space constraints, this effect is discussed in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.180.

4. The Pumsb data set, which is characterized by a highly dense upper part of the tree and a rather sparse bottom part, is an exception. Its I-Tree size is larger due to the overhead yielded by storing IMine control information for the bottom part of the tree.
compare the performance of the approaches on the extraction phase, the time for writing the generated item sets is not included. This time is comparable for all approaches.

Fig. 6 compares the FP-based algorithm with the Prefix-tree [17] 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. For large synthetic data sets, the CPU time required by the FP-based algorithm is always smaller than that required by the Prefix-Tree. The FP-based algorithm adopts exactly the same extraction approach as FP-growth. For low supports, it is significantly faster than FP-growth. FP-growth did not terminate correctly the extraction task on the Kosarak data set in the considered support range.

The FP-based algorithm, albeit implemented into a relational DBMS, yields performance always comparable with and sometimes better than the other algorithms. This performance is mainly due to the optimized storage of correlated data in few blocks, which supports selective data loading in memory. Hence, I/O costs are limited and significantly more memory space is available for the extraction task. This effect is particularly relevant for low supports, because representing in memory a large portion of the data set may significantly reduce the space for the extraction task, hence causing more memory swaps.

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.

4.3 Memory Consumption

Fig. 8 reports the peak main memory required by the two IMine-based algorithms during item set extraction. For both algorithms, the PostgreSQL buffer cache (with default size 512 Kbytes) is included in the memory held by the process. The FP-based and LCM-based algorithms are compared with the Prefix-Tree and LCM v.2 algorithms, respectively. Kosarak (sparse) and Pumsb (dense) data sets are discussed as representative examples.

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. Instead, the Prefix-Tree algorithm stores in memory all the data needed for the whole extraction process. Memory consumption for Prefix-Tree increases when larger data sets or lower supports are considered, because more data are stored in memory. The slightly higher memory consumption shown on the Pumsb data set by the FP-based algorithm is due to the buffer cache size.

Since the LCM-based and LCM v.2 algorithms store in memory all data needed during the extraction process, both algorithms show the same peak main memory behavior. Also in this case, the slight difference between the two algorithms is due to PostgreSQL buffer cache.

We also evaluated the average and total memory allocated during the extraction process (not reported here for lack of space). For both IMine-based algorithms, average
and peak memory are close. Thus, memory consumption is rather constant in time. The difference in both average and total memory is more significant between the FP-based and the Prefix-Tree algorithms. Since the buffer cache size is kept constant, FP-based memory requirements are rather stable with decreasing support thresholds, while for Prefix-Tree they increase significantly. FP-growth needs at least five times more memory than the other approaches, because it keeps in memory the whole frequent subset of the data set during the entire mining process.

4.4 Effect of IMine Layered Organization

Fig. 9 reports the I-Tree node distribution in the three layers for the default index configuration ($K_{avg} = 1.2$ and $K_{sup} = 0$ or 0.005, see Section 4.1). The Top layer is larger for dense data sets, while the Middle layer is always rather small. Pumsb and T10I200P20D2M represent two extreme cases. Both data sets are dense, but Pumsb has a high number of very frequent items, while T10I200P20D2M has a high number of medium frequency items. Hence, the Top layer is large in Pumsb (51.5 percent) and very small in T10I200P20D2M (less than 1 percent). The size of the Bottom layer increases with the data set sparseness. We analyze the effect of the I-Tree layered organization in terms of hit rate, item set extraction time, and index creation time.

Effect of layer partitioning. We compared the default I-Tree configuration (three layers) with a single layer I-Tree organization in the following two cases. 1) Only Bottom layer. The paths are stored without any correlation analysis (i.e., vertically). 2) Only Top layer. Correlation analysis is performed on the entire I-Tree. Fig. 10 reports the percentage difference with the default case for both the FP-based and LCM-based approaches on Pumsb and Kosarak. The support threshold is 14.78 percent for Pumsb and 0.024 percent for Kosarak. The hit rate is always lower than in the default configuration (Fig. 10a), thus highlighting

5. FP-growth memory consumption is not shown in Fig. 8b to better show the peak main memory differences among the other considered approaches.
the usefulness of both path partitioning and correlation analysis. However, path correlation by itself (case 2) is not beneficial. In general, no correlation (case 1) is worse than global correlation (case 2) except for one case (Kosarak). Since a decrease in hit rate corresponds to a larger number of I/O requests, it causes an increase in execution time (Fig. 10b). As expected, index creation time (Fig. 10c) is always lower without correlation analysis. Extensive correlation analysis takes a significantly larger time, especially for sparse data sets which are usually characterized by long and weakly correlated paths (e.g., Kosarak).

Varying the size of the Top layer. Parameter $K_{avg}$ selects the items in the Top layer. It ranges from 1 (all items are included) to $\#\text{Trans}$ (only items in all transactions are included, i.e., usually no item is included). The effect of $K_{avg}$ is analyzed on the T15I100P20C1D5M data set by considering the range $[1.1, \#\text{Trans}/10.000]$. The extreme values (1 and $\#\text{Trans}$) are not feasible for this data set, because they would require correlating (almost) the entire I-Tree. Fig. 11a plots the CPU time for the FP-based algorithm when varying the support. Any value in the range $[1.1-1.3]$ reduces the CPU time compared to the worst case ($K_{avg} = \#\text{Trans}/10.000$). The effect is more evident for low supports. When $K_{avg} = \#\text{Trans}/10.000$, the I-Tree Top layer includes a small subsets of nodes. Hence, long correlated paths are broken and stored in different physical blocks. During the mining process, I/O access time, and thus extraction time, increases. For this data set (and also for the other data sets, not reported here), the best index configuration is $K_{avg} = 1.2$. Similar considerations hold for the LCM-based algorithm, plotted in Fig. 11b. However, since the LCM-based algorithm exploits correlation analysis to a lesser extent, the effect of varying $K_{avg}$ is less evident.

4.5 Effect of Directly Writing Items on Disk

Direct writing only occurs when the entire I-Tree does not fit in main memory. Parameter $K_{Sup}$ allows selecting items which should be directly written on disk. $K_{Sup}$ ranges in the interval $[0,1]$. For $K_{Sup} = 0$, no item is directly written on disk (default configuration). For $K_{Sup} = 1$, all items with support lower than the average item support $\mu_{Sup}$ are directly written on disk. The analysis is performed on Kosarak ($\mu_{Sup} = 194$), T10I200P20D2M ($\mu_{Sup} = 464$), and T15I100P20C1D5M ($\mu_{Sup} = 2.423$). Fig. 12 reports the increase in the (overall) index size with respect to the default configuration. $K_{Sup}$ values in the range $[0.025, 0.075]$ yield a low increase (less than 0.8 percent) in index size. For $K_{Sup}$ in the range $[0.25, 1]$, the percentage variation in index size is mainly dependent on the data distribution. For Kosarak (the sparsest data set), the index size in the worst case ($K_{Sup} = 1$) increases by...
1.7 percent, while for T10I200P20D2M (the densest data set), it doubles with respect to the default configuration. In dense data sets, many transactions are represented by a single I-Tree path whose nodes have support larger than 1. By directly (i.e., separately) writing the corresponding transactions on disk, the I-Tree compact representation is lost, thus increasing the index size.

### 4.6 Scalability

The scalability of the FP-based and LCM-based algorithms has been analyzed by varying 1) the number of transactions and 2) the pattern length. All data sets were generated by means of the IBM generator [23]. All IMine indices have been created with the values $K_{avg} = 1.2$ and $K_{Sup} = 0.05$. The experiments in this section have been performed on a dual-processor quad-core Intel Xeon Processor E5430 with 8-Gbyte main memory running Linux kernel v. 2.6.17-10-generic.

To analyze the scalability of the IMine-based algorithms with respect to the number of transactions we considered large databases of size ranging from 10M (million) (database size 1.8 Gbytes) to 40M (8 Gbytes) transactions with an average length of 12 items. These data sets have 100,000 items, pattern length 10, and correlation grade 0.9. Fig. 13 plots the extraction time for various supports. Both algorithms scale well even for very large data sets. Since the number of extracted item sets grows significantly for very low supports (i.e., 0.1 percent), the process becomes computationally more expensive. However, the overall CPU time is still low, less than 1,000 seconds for the lowest considered support.

To analyze the scalability with respect to the pattern length, we considered transactional databases with 10M transactions and pattern length ranging from 10 (database size 1.8 Gbytes) to 20 (3 Gbytes), with 100,000 items and correlation grade 0.9. Fig. 14 plots the extraction time for the IMine-based algorithms with different support thresholds. By increasing the pattern length, the data set becomes denser. Hence, the I-Tree compactness increases. The extraction time is linear for a broad support range. As expected, correlation is beneficial for long pattern. For very low supports, the extraction time increases, but still to a limited extent (less than 600 seconds for the lowest support).

5 **RELATED WORK**

The wide diffusion of data warehouses caused an increased interest in coupling the item set extraction task with relational DBMSs. Different levels of coupling were proposed, ranging from loose coupling, given by SQL statements exploiting a traditional cursor interface [24], to tight coupling, which proposed various kinds of optimizations in managing the interaction with the DBMS [25], [26]. A parallel effort was devoted to the definition of expressive query languages, such as DMQL [27] or MINE RULE [28], to specify mining requests. These languages often proposed SQL extensions that support the specification of various types of constraints on the knowledge to be extracted (see [29] for a review of the most important languages). Chaudhuri et al. [30] perform one step further toward a tighter integration by proposing techniques to optimize queries including mining predicates. Knowledge of the mining model is exploited to derive from mining predicates simpler predicate expressions that can be exploited for access path selection like traditional database predicates.

Different disk-based data structures for frequent item set extraction from flat file have been proposed [7], [8], [9]. Ramesh et al. [9] propose B+tree-based indices to access data stored by means of either a vertical or a horizontal data representation. Item set extraction is performed by means of both APRIORI [1] and Eclat [31]. However, performance is usually worse than, or at best comparable to, flat file mining. In [32], an index structure based on signature files is proposed to drive the generation of candidate frequent item sets. However, to check the actual candidate frequency, the data set needs to be further accessed. More
recently, El-Hajj and Zaiane [7] stored the data set in a disk-based Inverted Matrix, which is then accessed to extract frequent item sets. This data structure is specifically suited for very sparse data sets, characterized by a significant number of items with unitary support. In [8], large databases are materialized on disk in different projected databases whose dimension fits in main memory. During the mining process, the projected database associated to each frequent item needs to be loaded in main memory. Hence, this approach requires several (costly) accesses to the potentially large number of projected data sets. Different from previous approaches, the IMine index is a disk-based data structure which supports the tight integration of item set extraction in a relational DBMS. The index structure is able to cope with both sparse and dense data distributions. The idea of tightly integrating item set mining with a relational DBMS was first introduced in [33]. The IMine index significantly enhances the index structure proposed in [33], by providing a more general approach, exploiting different extraction algorithms, and richer query options with respect to [33]. Furthermore, since the creation phase in [33] required to build in main memory the entire index before writing it on disk, its capability of mining and manipulating large data sets was rather limited.

A complementary approach is proposed in [34], where frequent closed item sets are stored in a disk-based structure named CFP-tree and algorithms have been proposed to retrieve patterns from there. However, for low supports, the solution set becomes too huge to be represented by means of the CFP-tree. The IMine index, instead, completely represents the entire data set, thus supporting item constrained extraction even without any support threshold.

Similarly to the FP-tree [3], the I-Tree is a prefix-tree structure. The I-Tree contains supplementary information to support more flexible data access, like brother node pointers (not available in the FP-tree). The I-Tree is a disk resident data structure which provides a complete representation of a data set. It is built once and exploited for many extraction sessions, possibly with different support thresholds and item constraints. By contrast, the FP-tree is a memory resident data structure, which is generated for every extraction session. For large data sets or when low supports are considered, the entire FP-tree may not fit in memory. The problem of the size of the complete FP-tree in memory was addressed in [3] by proposing to load only projected data sets. The number of projected data sets may become too large and selecting the best subset to store is a difficult task. The IMine index encodes in a unique structure all these projected data sets. Furthermore, any projected data set can be selectively loaded by accessing the appropriate index blocks.

Various approaches have been proposed to enforce constraints into the item set extraction process [10], [11], [12], [13], [15]. Constraints have been classified as anti-monotonic, monotonic, succinct, convertible, and inconvertible [15]. The first four constraint types (and a combination of them) can be pushed deep into the frequent pattern growth algorithm. The approach proposed in [15] can be straightforwardly supported by the IMine index by means of the available access methods, as discussed in Section 3.2. Since the approach proposed in [15] is based on FP-growth, it can suffer the same shortcomings in case of large databases or low supports (mainly, very high memory consumption). The IMine-based approach, instead, allows selectively loading the data set portion needed by the current extraction process, thus overcoming these limitations.

6 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


Elena Baralis received the master’s degree in electrical engineering and the PhD degree in computer engineering from Politecnico di Torino, Torino, Italy. She has been a full professor in the Dipartimento di Automatica e Informatica, Politecnico di Torino since January 2005. Her current research interests include the field of databases, in particular data mining, sensor databases, and bioinformatics. She has published numerous papers on journal and conference proceedings. She has managed several Italian and EU research projects.

Tania Cerquitelli received the master’s degree in computer engineering and the PhD degree from the Politecnico di Torino, Torino, Italy, and the master’s degree in computer science from the Universidad De Las Americas Puebla. She has been a postdoctoral researcher in computer engineering in the Dipartimento di Automatica e Informatica, Politecnico di Torino since January 2007. Her research interests include the integration of data mining algorithms into database system kernels and sensor databases.

Silvia Chiusano received the master’s degree and the PhD degree in computer engineering from Politecnico di Torino, Torino, Italy. She has been an assistant professor in the Dipartimento di Automatica e Informatica, Politecnico di Torino since January 2004. Her current research interests include the areas of data mining and database systems, in particular integration of data mining techniques into relational DBMSs, and classification of structured and sequential data.