Constrained itemset mining
on a sequence of incoming data blocks

Elena Baralis, Tania Cerquitelli, Silvia Chiusano
Politecnico di Torino - Italy
{elena.baralis, tania.cerquitelli, silvia.chiusano}@polito.it

Abstract

Many real-life databases are updated by means of incoming business information. In these databases (e.g., transactional data from large retail chains, call-detail records) the content evolves through periodical insertions (or deletions) of data blocks. Since data evolve over time, algorithms have to be devised to incrementally update data mining models.

This paper presents a novel index, called I-Forest, to support itemset mining on incoming data blocks, where new blocks are inserted periodically, or old blocks are discarded. The I-Forest structure provides a complete data representation and allows different kind of analyses (e.g., investigate quarterly data), besides supporting user-defined time and support constraints.

The I-Forest index has been implemented into the PostgreSQL open source DBMS and exploits its physical level access methods. Experiments, run for both sparse and dense data distributions, show the effectiveness of the I-Forest based approach to perform itemset mining with both time and support constraints. The execution time of the I-Forest based itemset mining technique is often faster than the Prefix-Tree algorithm accessing static data on flat files.

1 Introduction

Data mining is a relatively new research field that is focused on automatic extraction of useful information hidden in huge databases. Since many real-life databases (e.g., data marts) are updated by means of blocks of periodically incoming business information, data mining algorithms face new challenges. Consider for example a database which is periodically updated through either additions of new transaction blocks or deletions of obsolete ones. Examples of evolving datasets are transactional data from large retail chains, web server logs, financial stock tickers, and call-detail records. Data can be described as a sequence of incoming data blocks, where new blocks arrive periodically [13, 14]. Since data evolve over time, algorithms have to be devised to incrementally update data mining models. Different kinds of analysis could be performed over such data like (i) mining all available data (ii) mining only the most recent data (e.g., last month data), (iii) mining periodical data (e.g., quarterly data) and (iv) mining selected data blocks (e.g., data related to the first month of last year and the first month of this year). By analyzing the extracted knowledge (e.g., correlation among sold items expressed by means of association rules), marketers are able to identify users’ preferences and exploit this information for marketing purposes.
A well-known data mining problem is association rule mining [2]. Association rule mining is a two-step process: (i) Frequent itemset extraction and (ii) association rule generation from frequent itemsets. Research activity in association rule mining is mainly focused on defining efficient itemset extraction algorithms [2, 4, 20, 25, 28, 32, 33]. The data to be analyzed is (possibly) extracted from the DBMS and stored into binary files (i.e., flat files). Although algorithms for frequent itemset mining have been studied exhaustively (see [15, 17]), less interest has been devoted to the implementation of frequent itemset algorithms in and for relational database management systems. However, the demand for integration of data mining tools into the existing database management systems is significant. Coupling with database systems has been at best loose, and access to data in a DBMS has been usually provided through an ODBC or SQL cursor interface [3, 21, 22]. However, no attempt to achieve a complete integration of efficient external structures performing mining activities into a true relational DBMS has been proposed so far. DBMSs exploit indices to improve performance on complex queries. The intuition that the same approach could be “exported” to the data mining domain is the driving force of this paper. A true integration of a novel data mining index into the PostgreSQL open source DBMS is presented, highlighting advantages and disadvantages of the proposed disk-based data structures. DBMS kernel primitives have been designed and implemented to materialize the index structure on disk and to enhance itemset mining on evolving databases.

This paper presents a hierarchical index structure, called Itemset-Forest (I-Forest), for mining data modeled as sequences of incoming data blocks. Users can refine the extraction task by specifying (i) time constraints (i.e., the temporal period they are interested in) and (ii) support constraints (i.e., minimum itemset frequency). Since the I-Forest index includes all the information potentially needed during the extraction task, it is a covering index. Hence, itemset mining can be performed by means of the index alone, without accessing the database.

The most notable features of the proposed index are: (i) **Two levels of indexing** to efficiently access interesting data useful for the current mining process. The first level allows a selective access to the (physical) index blocks for loading the database projection (i.e., set of data blocks satisfying the constraints) in main memory. The second level stores the complete representation of the evolving database, as no support threshold is enforced during the index creation phase. Hence, the proposed index can be easily exploited for itemset mining with any support and time constraints. (ii) **Efficient index updating.** The I-Forest index is characterized by a set of compact structures, one for each incoming data block. Each structure provides a compact local representation of the data block. Hence, this flexible structure allows dynamic updating of the I-Forest index when new data blocks are inserted or old data blocks are discarded. (iii) **Ability to model different data distributions.** An interesting feature of the index is its ability to represent distinct data blocks by means of different data structures. Hence, the data structure of a block can be easily adapted to different data distributions. (iv) **Efficient information access.** The physical organization of the I-Forest index supports efficient data access to load in main memory only the data required for the current itemset extraction task. The I-Forest index allows selective access to the required information, thus reducing disk access overhead during the extraction task.

The index has been implemented into the open source PostgreSQL DBMS [27] and exploits its access level methods. Our approach, albeit implemented into a relational DBMS, yields performance comparable to and, for low support thresholds, better than the Prefix-tree algorithm [18] (FIMI ’03 Best Implementation) accessing static data on flat file.

The paper is organized as follows. Section 2 defines the problem addressed in this paper. Section 3 describes the structure of the proposed index, while Section 4 presents an algorithm
to efficiently extract itemsets enforcing time and support constraints. Section 5 discusses the experiments performed to evaluate the effectiveness of the proposed index. Section 6 compares our approach with related work. Finally, Section 7 draws conclusions and presents future developments of the proposed approach.

2 Problem statement

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of items. A transactional data block $b$ is a collection of transactions, where each transaction $T$ is a set of items in $\mathcal{I}$. Let $b_k$ be the instance of $b$ arrived at time $k$, where $k$ is a time identifier. A transactional database $\mathcal{D}$ at time $t$ is a finite sequence of data blocks $\{b_1, b_2, \ldots, b_t\}$ denoted as $\mathcal{D}[1..t]$.

$\mathcal{D}[1..t]$ can be represented as a relation $R$, where each tuple is a triplet $(\text{Time}, \text{TransactionId}, \text{ItemId})$. When a new block is inserted, its transactions are added to $R$ and $t$ is updated. Analysis can be performed on sets of (possibly) non-consecutive blocks. We denote as $\Omega$ the set of time identifiers of the analyzed blocks (e.g., $\Omega = \{1, 2, 5\}$ means that we are interested in blocks $\{b_1, b_2, b_5\}$). $R_\Omega \subseteq R$ is the set of tuples associated to the blocks in $\Omega$.

Given a set of constraints $\psi$, the itemset extraction task mines the complete set of itemsets in $R$ which satisfy $\psi$. Constraints in $\psi$ are among the following:

- Time constraint, which selects a subset of blocks in $R$ by means of $\Omega$.
- Support constraint, which defines the minimum support threshold to perform itemset extraction.

3 The I-Forest Index

I-Forest is a hierarchical index suitable to efficiently perform itemset mining on evolving databases, i.e., databases whose content changes over time through periodical insertions (or deletions) of data blocks (e.g., transactional data from large retail chains). By means of the I-Forest index, incoming data blocks are stored on disk using appropriate (compact) structures. During the mining phase, only the data required by the current mining process is actually loaded in main memory. Furthermore, the I-Forest structure can be incrementally updated by considering both incoming and removed data blocks.

To allow different kinds of analysis and easy incremental insertion of new blocks (or deletion of obsolete ones), each block is represented separately and independently of all others. Hence, the index is a forest of structures that represent data blocks in $R$. Each data block $b_k$ in $R$ is stored using a compact structure named Itemset Forest-block (IF-block). A different structure, the Itemset Forest-Btree (IF-Btree), is used to link information belonging to different blocks that are accessed together during the mining process. The IF-Btree is a B+Tree structure, which provides selective access to data in the IF-blocks to support efficient retrieval of information. The complete structure of the I-Forest is shown in Figure 1.

Since the I-Forest index is designed to be exploited for many extraction sessions with different constraints, it gathers information on all the available data. No constraint (e.g., support) is enforced during the index creation phase. Hence, I-Forest is a covering index, i.e., it allows itemset extraction without accessing relation $R$. Its data structures include both the time and the item identifier attributes. During the extraction phase, time and/or support constraints may be enforced to refine the itemset extraction task.
A notable feature of the proposed approach is the independency of its components. The IF-Btree structure may be implemented by means of any kind of representation for storing sets of transactions (e.g., prefix-tree [20], array lists [34], Inverted Matrices [11], Patricia-Tries [29]). Since each block $b_k$ is stored independently by means of an IF-block, which provides a lossless and compact representation of the corresponding portion of $R$, each IF-block may adopt a different data representation. Hence, the I-Forest index can easily suit skewed block distributions.

Furthermore, the I-Forest index can deal with any large evolving database. A remarkable feature of the block partitioning approach (i.e., block evolution [14]) is its ability to deal with databases of significant size by partitioning them in smaller, more manageable chunks without incurring a significant overhead due to partitioning (for further discussion of this issue, see Section 5.2).

### 3.1 I-Forest structure

The I-Forest index includes two elements: The Itemset Forest-blocks (IF-blocks) and the Itemset Forest-Btree (IF-Btree). In the following we describe each structure in detail.

**IF-blocks.** Many different compact structures could be adopted for representing IF-blocks (e.g., FP-tree [20], Inverted Matrix [11], Patricia-Tries [29], I-Tree [7]). Currently, each IF-block is represented by means of a slight variation of the I-Tree. For each (relational) data block $b_k$ in $R$, an I-Tree ($I_{t_k}$) and an Item List ($I_k$) are built [7]. The $I_{t_k}$ associated to $b_k$ is a prefix tree, where each node corresponds to an item and each path encodes one or more transactions in $b_k$. Each node is associated with a node support value. This value is the number of transactions in $b_k$ which contain all the items included in the subpath reaching the node. Each item is associated to one or more nodes in the same $I_{t_k}$.

$I_k$ has one entry for each item in $b_k$, for which it reports the block$_k$-item-support value, i.e., the item frequency in $b_k$, and the block identifier $k$. The block$_k$-item-support value is obtained by adding the supports of all nodes in $I_{t_k}$ associated to the item. The global item support value is the frequency of the item in $R$. This value is obtained by adding the block$_k$-item-supports of the item for each block $b_k$.

Figures 2 and 3 report (in a more succinct form than the actual relational representation)
two data blocks, inserted at times \( t = 1, 2 \), used as a running example. Figures 4 and 5 show the structure of the corresponding \( I_1 \) and \( I_2 \) associated to the blocks 1 and 2 respectively. The corresponding \( I_1 \) and \( I_2 \) are omitted to ease readability. Consider item \( e \) in \( I_1 \). Its \( block_1 \)-item-support is 9 (there are two nodes associated with item \( e \), the first has node support 6 and the second 3). Furthermore, global item support of \( e \) is 13 (its \( block_1 \)-item-support is 9 and its \( block_2 \)-item-support is 4).

As shown in Figures 4 and 5, in each \( I_{tk} \) nodes are clustered in three layers: top, middle, and bottom [7]. Correlation analysis is exploited to store nodes accessed together during the mining process in the same disk block, thus reducing the number of reads.

Nodes in the IF-block are linked by means of pointers which allow the retrieval of the index portion required by the mining task. Each node \( N_i \) has pointers to three index blocks: (i) The physical location (i.e., disk block and offset) of its parent (continuous edges in Figures 4 and 5), (ii) the physical location of its first child (dashed edges in Figures 4 and 5), and (iii) the physical location of its right brother (dotted edges in Figures 4 and 5). The pointers allow both bottom-up and top-down tree traversal, thus enabling itemset extraction with both time and support constraints.

**IF-Btree.** It is a B+Tree structure which allows selective access to the IF-block disk blocks during the mining process. It has one entry for each item in relation \( R \). The IF-Btree leaves contain pointers to all nodes in the IF-blocks. Each pointer contains the node physical address and the identifier of the data block of the node itself. This data organization enables to
selectively load only the required portions of the IF-blocks for the current extraction session.

3.2 Updating the I-Forest index

The proposed index supports efficient index updating when data blocks are inserted or deleted. When a new data block \( b_k \) is available, the corresponding IF-block \( (I_k) \) is built. The IF-Btree structure is also updated by inserting pointers to the nodes in \( I_k \). Hence, \( I_k \) data are linked to the old data in the I-Forest index. When a data block \( b_k \) is discarded, the corresponding \( I_k \) is removed and the IF-Btree is updated by removing pointers to the nodes in \( I_k \). Functions available in the PostgreSQL library are used to materialize on disk each \( I_k \) and to update the IF-Btree.

3.3 Index Storage

The I-Forest index is stored in two relational tables. Table \( T_{\text{Items}} \) stores all \( I_k \), while table \( T_{\text{IF-blocks}} \) stores every \( I_k \). The IF-Btree is stored in a B+Tree structure. To access records in \( T_{\text{IF-blocks}}, T_{\text{Items}} \), and in the IF-Btree, functions available in the access methods of PostgreSQL [27] are used. Table \( T_{\text{Items}} \) contains one record for each item in each \( I_k \). Each record contains the data block identifier, the item identifier, and \( \text{block}_k \)-item-support. Table \( T_{\text{IF-blocks}} \) contains one record for each node in each \( I_k \). Each record contains the node identifier, the item identifier, the local node support, the physical location (i.e., block number and offset within the block) of its parent, the physical location of its first child, and the physical location of its right brother.

4 Mining Itemsets

This section describes how itemset extraction with different constraints exploits the I-Forest index. Users may refine their mining queries by means of two different constraints. (i) \textit{Time constraint}. Set \( \Omega \) contains the time identifiers of the selected data blocks. When the time identifier \( k \) is included in \( \Omega \), the corresponding data block \( b_k \) is analyzed. When no time constraint is enforced, all data blocks are considered in the extraction process. (ii) \textit{Support constraint}. Itemsets are extracted when their frequency is higher than a minimum threshold (\textit{MinSup}). Itemset extraction is performed in two steps. First, eligible items, i.e., items which satisfy all the enforced constraints, are selected. Then, the extraction process is performed. To retrieve all the necessary information, both tables \( T_{\text{IF-blocks}} \) and \( T_{\text{Items}} \), and the IF-Btree are accessed using the read functions available in the PostgreSQL access methods [27].

4.1 Enforcing constraints

The constraint enforcement step selects the items which belong to blocks in \( \Omega \) and satisfy the support constraint. These items are stored in set \( \Lambda \). Algorithm 1 shows the pseudo code of the constraint enforcement algorithm. It performs a sequential access to records in table \( T_{\text{Items}} \). When no support constraint is enforced, items are inserted into \( \Lambda \) if they belong to at least one block in \( \Omega \) (lines 6-8). When a support constraint is enforced, we compute the \textit{joint support} for each item by adding the current \( \text{block}_k \)-item-supports for all blocks in \( \Omega \) (lines 9-11). Items with joint support lower than the support threshold are finally pruned from \( \Lambda \) (lines 15-17).
Algorithm 1 Constraint enforcement algorithm

Require: $T_{items}, \Omega, MinSup$
Ensure: $\Lambda$ set of eligible items

1: $\Lambda = \emptyset$
2: Init_Seq_Scan($T_{items}$) // Sequential scan of table $T_{items}$
3: Record=Get.Next_Record($T_{items}$)
4: repeat
5: if Record.BlockId $\in \Omega$ then
6: if Record.Item $\notin \Lambda$ then
7: Insert_Item(Record.Item,$\Lambda$) // insert item into $\Lambda$
8: end if
9: if $MinSup > 0$ then
10: Update_Joint_Support($\Lambda$,Record) // update joint support for the item
11: end if
12: end if
13: Record=Get.Next_Record($T_{items}$) // read next record from table $T_{items}$
14: until Record IS NOT NULL
15: if $MinSup > 0$ then
16: Prune_Unfrequent_Items($\Lambda$,MinSup);
17: end if
18: return $\Lambda$

4.2 Itemset extraction

Itemset extraction (see Algorithm 2) is performed in two steps: (i) Retrieval of the necessary data from the I-Forest index (lines 4-14) and (ii) itemset extraction from the loaded data (lines 16-17). The proposed extraction algorithm is briefly described in the following.

For each item in $\Lambda$, all the I-Forest paths including the item are read. To this aim, the IF-Btree leaves storing the pointers to all I-Forest nodes associated to the item (line 5) are retrieved. Each pointer includes the node physical address and the identifier of the data block of the node itself. Only nodes belonging to blocks in set $\Omega$ are considered (line 8-12).

For each node associated to the item, all the I-Forest paths including it are retrieved from disk (lines 9-11). Each node is read from table $T_{IF-blocks}$ by means of its physical address (line 9). The path between the node and the tree root is traversed bottom up. This path is read from table $T_{IF-blocks}$ by following the physical address of the parent node (line 10). The paths in the node subtree are traversed top down. These paths are read from table $T_{IF-blocks}$ by following the physical addresses of the first child nodes and the right brother nodes (line 11).

Once read, paths are initially stored in main memory in a temporary data structure, called $D_p$. When all paths including the current item have been retrieved, data structure $D_p$ is compacted into a tree structure similar to the FP-Tree [20] (line 16). Next, itemset extraction is performed by means of an adaptation of the FP-Growth algorithm [20] (line 17).
Algorithm 2 Itemset extraction algorithm

Require: $T_{IF-blocks}, IF-Btree, \Omega, MinSup, \Lambda$

Ensure: $Itemset\#$ number of extracted itemsets

1: $Itemset\# = 0$
2: for all items in $\Lambda$ do
3:     // Step 1: Data retrieval from disk
4:     $D_p = \emptyset$
5:     Leaves = Get_IF-Btree_Leaves(Item, IF-Btree)
6:     NodePointer = Get_Next_NodePointer(Item, Leaves)
7:     repeat
8:         if NodePointer.BlockId $\in \Omega$ then
9:             Node = Read_Node($T_{IF-blocks}, NodePointer.PhyAddr$)
10:            Reach_Root($T_{IF-blocks}, Node, D_p$)
11:            Visit_Subtree($T_{IF-blocks}, Node, D_p$)
12:         end if
13:         NodePointer = Get_Next_NodePointer(Item, Leaves)
14:     until NodePointer IS NOT NULL
15:     // Step 2: Itemset extraction
16:     $cD_p = BuildCompactTree(D_p)$
17:     Mine($cD_p, MinSup, Itemset\#$)
18: end for
19: return $Itemset\#$

5 Experimental Results

We validated our approach by means of several experiments addressing the following issues: (i) Execution time for the I-Forest index creation and index size, (ii) execution time and memory usage for itemset extraction with time and support constraints, (iii) effect of the DBMS buffer cache size on hit rate, and (iv) effect of dataset partitioning.

We run experiments by considering both dense and sparse data distributions. We report three representative examples. Two datasets (Connect and Pumsb [12]) are very dense. The third dataset (Kosarak [12]) is a large and sparse dataset. To simulate block evolution, we considered three different configurations. For the first two, denoted as Dataset-50+50 and Dataset-25+25+25+25, we have split the original dataset in two and four blocks respectively with the same number of transactions. The last configuration, denoted as Dataset-100+100, is composed by two identical blocks obtained by cloning the original dataset. The original dataset, denoted as Dataset-100, represents our lower bound on performance since it has no overhead due to block partitioning. To validate our approach, we compare our performance to the FIMI best implementation algorithm Prefix-tree [18], a very effective state of the art algorithm for frequent itemset extraction from flat file.

To analyze the performance of the extraction process in large evolving databases, we considered two synthetic datasets characterized by different data distributions and number of transactions. The datasets were generated by means of the IBM generator [1]. We set the number of transactions $D=2M$ and $10M$, number of distinct items $I=200K$, pattern length $P=10$ and 15, average length of transaction $T=10$, and the default setting for all other parameters. Dataset
Table 1: Characteristics of dataset configurations and corresponding indices

<table>
<thead>
<tr>
<th>Dataset configuration</th>
<th>Transactions</th>
<th>Items</th>
<th>AvTrSz</th>
<th>Size(KB)</th>
<th>IF-blocks (KB)</th>
<th>IF-Btree (KB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect-100</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>Connect-100+100</td>
<td>135,114</td>
<td>129</td>
<td>43</td>
<td>51,054</td>
<td>45,268</td>
<td>8,420</td>
<td>22.10</td>
</tr>
<tr>
<td>Connect-50</td>
<td>33,779</td>
<td>129</td>
<td>43</td>
<td>12,002</td>
<td>12,138</td>
<td>-</td>
<td>7.50</td>
</tr>
<tr>
<td>Connect-50+100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>24,662</td>
<td>4,588</td>
<td>14.70</td>
</tr>
<tr>
<td>Connect-25</td>
<td>16,890</td>
<td>129</td>
<td>43</td>
<td>6,497</td>
<td>6,210</td>
<td>-</td>
<td>2.67</td>
</tr>
<tr>
<td>Connect-25+25+25+25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25,215</td>
<td>4,691</td>
<td>10.63</td>
</tr>
<tr>
<td>Pumsb-100</td>
<td>98,092</td>
<td>2,144</td>
<td>37.01</td>
<td>35,829</td>
<td>37,932</td>
<td>10,789</td>
<td>34.47</td>
</tr>
<tr>
<td>Pumsb-100+100</td>
<td>196,184</td>
<td>2,144</td>
<td>37.01</td>
<td>71,658</td>
<td>115,932</td>
<td>21,562</td>
<td>69.00</td>
</tr>
<tr>
<td>Pumsb-50</td>
<td>49,046</td>
<td>1946</td>
<td>37.01</td>
<td>17,712</td>
<td>30,415</td>
<td>-</td>
<td>21.50</td>
</tr>
<tr>
<td>Pumsb-50+50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>62,441</td>
<td>11,625</td>
<td>45.68</td>
</tr>
<tr>
<td>Pumsb-25</td>
<td>24,523</td>
<td>1787</td>
<td>37.01</td>
<td>8,756</td>
<td>16,668</td>
<td>-</td>
<td>8.41</td>
</tr>
<tr>
<td>Pumsb-25+25+25+25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>68,132</td>
<td>12,676</td>
<td>34.74</td>
</tr>
<tr>
<td>Kosarak-100</td>
<td>1,017,029</td>
<td>41,244</td>
<td>7.9</td>
<td>85,435</td>
<td>312,647</td>
<td>55,401</td>
<td>893.81</td>
</tr>
<tr>
<td>Kosarak-100+100</td>
<td>2,034,058</td>
<td>41,244</td>
<td>7.9</td>
<td>170,870</td>
<td>625,294</td>
<td>116,472</td>
<td>1787.50</td>
</tr>
<tr>
<td>Kosarak-50</td>
<td>508,515</td>
<td>35,586</td>
<td>7.9</td>
<td>42,310</td>
<td>160,020</td>
<td>-</td>
<td>648.56</td>
</tr>
<tr>
<td>Kosarak-50+50</td>
<td>508,514</td>
<td>35,139</td>
<td>7.9</td>
<td>43,125</td>
<td>159,458</td>
<td>-</td>
<td>651.06</td>
</tr>
<tr>
<td>Kosarak-25</td>
<td>254,258</td>
<td>30,447</td>
<td>7.9</td>
<td>20,692</td>
<td>81,416</td>
<td>-</td>
<td>115.25</td>
</tr>
<tr>
<td>Kosarak-25+25+25+25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>326,552</td>
<td>60,386</td>
<td>445.08</td>
</tr>
</tbody>
</table>

names indicate parameter values and original datasets are denoted as Dataset-100. We split the T10I200P15D2M dataset in 2 blocks and the T10I200P10D10M dataset in 5 blocks. These configurations allowed us to analyze blocks with different number of transactions (i.e., 1,000,000 and 2,000,000 transactions for each block respectively). On these synthetic datasets we also analyzed the performance of the extraction process for increasing numbers of selected index blocks and for various support thresholds.

Both the index creation procedure and the itemset extraction algorithm are coded into the PostgreSQL open source DBMS [27]. They have been developed in ANSI C. Experiments have been performed on a 2800Mhz Pentium IV PC with 2Gbyte main memory. The buffer cache of PostgreSQL DBMS has been set to the default size of 64 blocks (block size is 8Kbyte). The reported execution times are real times, including both system and user times. They are obtained using the unix time command as in [16]. All indices have been generated without enforcing any support threshold.

5.1 I-Forest creation and structure

For the considered configurations of the Connect, Pumsb, and Kosarak datasets, Table 1 reports both the number of transactions and items, and the average transaction size (AvTrSz). Table 1 also shows the size of the corresponding I-Forest index. Table 2 reports the same information for T10I200P15D2M and T10I200P10D10M synthetic datasets. In this case datasets are partitioned in 2 and 5 blocks respectively (denoted as Dataset-b_i). Each block T10I200P15D2M-b_i is char-
Table 2: Characteristics of large dataset configurations and corresponding indices

<table>
<thead>
<tr>
<th>Dataset configuration</th>
<th>Transactions</th>
<th>Items</th>
<th>AvTrSz</th>
<th>Size (KB)</th>
<th>IF-blocks (KB)</th>
<th>IF-Btree (KB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I200P15D2M-100</td>
<td>2,000,000</td>
<td>75,633</td>
<td>16.09</td>
<td>437,248</td>
<td>605,323</td>
<td>112,672</td>
<td>930.25</td>
</tr>
<tr>
<td>T10I200P15D2M-b1</td>
<td>1,000,000</td>
<td>72,524</td>
<td>16.09</td>
<td>209,920</td>
<td>313,040</td>
<td>-</td>
<td>530.16</td>
</tr>
<tr>
<td>T10I200P15D2M-b2</td>
<td>1,000,000</td>
<td>72,536</td>
<td>16.09</td>
<td>227,328</td>
<td>313,040</td>
<td>-</td>
<td>542.43</td>
</tr>
<tr>
<td>T10I200P15D2M-b1-b2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>626,448</td>
<td>117,673</td>
<td>1,072.59</td>
</tr>
<tr>
<td>T10I200P10D10M-100</td>
<td>10,000,000</td>
<td>97,983</td>
<td>12.76</td>
<td>1,785,050</td>
<td>3,424,026</td>
<td>642,206</td>
<td>4,321</td>
</tr>
<tr>
<td>T10I200P10D10M-b1</td>
<td>2,000,000</td>
<td>55,633</td>
<td>12.76</td>
<td>345,950</td>
<td>723,832</td>
<td>-</td>
<td>902.85</td>
</tr>
<tr>
<td>T10I200P10D10M-b2</td>
<td>2,000,000</td>
<td>55,636</td>
<td>12.76</td>
<td>359,794</td>
<td>723,832</td>
<td>-</td>
<td>923.93</td>
</tr>
<tr>
<td>T10I200P10D10M-b3</td>
<td>2,000,000</td>
<td>55,635</td>
<td>12.76</td>
<td>359,848</td>
<td>723,832</td>
<td>-</td>
<td>933.65</td>
</tr>
<tr>
<td>T10I200P10D10M-b4</td>
<td>2,000,000</td>
<td>55,620</td>
<td>12.76</td>
<td>359,848</td>
<td>723,832</td>
<td>-</td>
<td>927.62</td>
</tr>
<tr>
<td>T10I200P10D10M-b5</td>
<td>2,000,000</td>
<td>55,630</td>
<td>12.75</td>
<td>359,848</td>
<td>723,832</td>
<td>-</td>
<td>1,006.06</td>
</tr>
<tr>
<td>T10I200P10D10M-b1-b5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3,624,176</td>
<td>676,419</td>
<td>4,694.11</td>
</tr>
</tbody>
</table>

characterized by 1,000,000 transactions, while each T10I200P10D10M-bi block contains 2,000,000 transactions.

For each dataset configuration, the overall size of the IF-blocks is obtained by summing the size of the IF-blocks in the I-Forest index. The IF-Btree contains pointers to all nodes in the IF-blocks, hence the IF-Btree size is proportional to the total number of nodes in the IF-blocks of the I-Forest.

The I-Tree internal block representation is more suitable for dense data distributions. In the I-Tree correlated transactions are represented by a single path. Hence, in dense datasets where data are highly correlated, the I-Tree structure provides good data compression. For sparse data distributions, where data are weakly correlated, a lower data compression is achieved. Storage of the IF-blocks requires more disk blocks. Alternative representations (e.g., Patricia-Tries [29]) may reduce the required disk space. Block partitioning increases the overall I-Forest size (including both IF-blocks and IF-Btree elements). The increase is higher for dense than for sparse data distributions, but it is never relevant. In dense datasets, the I-Tree compactness is partially lost due to block partitioning. Transactions, which were correlated in the original dataset and belong to different data blocks, are represented as disjoint paths. In sparse datasets, instead, many transactions are already characterized by disjoint paths in the original dataset.

Connect-25+25+25+25 and Connect-50+50 configurations require 11.36% and 8.9% more memory than Connect-100. Pumsb-25+25+25+25 and Pumsb-50+50 configurations require 11.7% and 7.7% more memory than Pumsb-100. Instead, for the Kosarak dataset, which is sparser, the Kosarak-25+25+25+25 and Kosarak-50+50 configurations require 2.16% and 4.28% more memory than Kosarak-100.

Index creation time is mainly given by the trade-off between two factors: Writing the index blocks on disk and performing correlation analysis. Due to block partitioning, a larger number of nodes has to be written. However, smaller data blocks reduce the complexity of the correlation analysis task, and consequently the required time. Results show that the index creation time for Dataset-25+25+25+25 configuration is always lower than (or at most comparable to) the time for Dataset-100.

Since T10I200P15D2M and T10I200P10D10M are very large and sparse datasets, block partitioning does not increase significantly the overall size of the I-Forest index. In these datasets most transactions are already represented by disjoint paths in the original dataset. The main overhead in the index size is due to the storage of the physical pointers that link nodes. This information is required to support the effective retrieval of correlated data during the mining process, thus reducing the number of disk reads.
5.2 Itemset extraction performance

In this section we analyze: (a) Frequent itemset extraction performance, (b) effect of dataset partitioning, and (c) performance of itemset extraction with both time and support constraints.

Frequent itemset extraction. To evaluate the performance of our approach, we considered the following configurations of the I-Forest index: (i) The best case, represented by Dataset-100 (no partitioning), (ii) general cases, represented by Dataset-50+50 and Dataset-25+25+25+25, and (iii) the worst case, represented by Dataset-100+100 (being the two data blocks identical, during the extraction phase both IF-blocks have always to be accessed).

Figures 6, 7, and 8 report the run time for frequent itemset extraction with various support thresholds. Figures 6, 7, and 8 also show the run time for the Prefix-Tre algorithm [18] on Dataset-100 (flat file). Since our objective is to compare the performance of our approach on the extraction phase, the time required for writing the generated itemsets is not included. This time is comparable for both approaches. When low support thresholds are considered, our approach, albeit implemented into a relational DBMS, is always significantly faster than the Prefix-tree algorithm on flat file. When higher supports are considered, our approach often yields performance comparable to the Prefix-tree algorithm. For T10I200P10D10M dataset (see Figure 7(b)) our approach outperforms the Prefix-Tree for any considered support thresholds. Furthermore, when using the I-Forest index based approach, the overhead due to data retrieval from disk is counterbalanced by an efficient memory usage during the extraction phase. Memory usage is further discussed in the next section.

Effect of dataset partitioning. Figures 9 and 10, which are a zoomed version of the former figures, analyze the overhead on frequent itemset extraction introduced by block partitioning for Connect, Pumsb and Kosarak datasets in the four configurations. Block partitioning introduces an overhead on data retrieval, because data in different IF-blocks are accessed. Experimental results show that this overhead does not significantly affect the extraction performance for any support threshold. In particular, the overhead due to block partitioning is expected to (slightly) increase when mining itemsets with high support constraints. Items occurring in many transactions are potentially correlated and may be represented in the same index path. Block
partitioning may split this path into different blocks, thus affecting data retrieval performance. On the other hand, when dealing with smaller data blocks, the index creation procedure can provide a clever physical organization of each IF-block, because correlation analysis may become more effective. Hence, within each IF-block, data accessed together during the extraction phase are actually stored in the same (disk) block, thus reducing the I/O cost. When lower supports are considered, weakly correlated items are analyzed. These items are naturally represented in disjoint index paths.

For the configuration Dataset-100+100, the run time for frequent itemset extraction is higher than for all the other configurations. Recall that Dataset-100+100 is obtained by cloning the original dataset. Hence, to retrieve the index data, the number of disk accesses is doubled with respect to the configuration Dataset-100.

Itemset extraction with time and support constraints. To analyze the performance of the extraction process when increasing the number of selected index blocks and for various support thresholds, we ran experiments enforcing both time and support constraints. Experiments have been performed on both large synthetic datasets T10I200P15D2M and T10I200P10D10M. We considered different time constraints, with an increasing number of blocks to be mined (Ω lists the blocks to be mined). For each time constraint, Figure 11 reports the run time of itemset extraction.
extraction by varying the support threshold. The extraction time significantly grows with the number of analyzed blocks. The increase is mainly due to the cost of accessing data in different index blocks. The increment is more evident for lower supports. Being the original dataset sparse, most of the items are characterized by a low frequency. Hence, a large amount of data has to be retrieved when low support thresholds are considered.

5.3 Memory consumption

We compared the memory consumption$^2$ of our approach with the Prefix-Tree algorithm on flat file. We report the average memory (Figure 12(a)) and the total memory (Figure 12(b)) required during the extraction process. For the I-Forest index the buffer cache of the DBMS (default size of 64 blocks) is included in the memory held by the current process. The Kosarak dataset is discussed as a representative example.

Figure 12(a) shows that Prefix-Tree, on average, needs a significantly larger amount of memory than our index based approach. It requires an amount of overall memory (Figure 12(b)) smaller than I-Forest only for high support values. The I-Forest index allows the extraction algorithm to selectively load in memory only the data required for the current extraction phase.
Hence, both average and total required memory may become significantly smaller than that re-
quired by Prefix-Tree, which stores the entire data structure in memory for the whole extraction
process. Since I-Forest has a better memory management, a larger memory space is available
for the extraction process, thus avoiding the occurrence of memory swaps or the lack of mem-
ory space. Furthermore, since the size of the PostgreSQL DBMS buffer cache is kept constant
(see Section 5.4 for a discussion of this issue), global and average memory requirements of our
approach are rather stable with decreasing support thresholds.

5.4 Buffer cache size

The hit rate is the ratio between the number of hits (accessed blocks already available in the
buffer cache) and the total number of accesses to index blocks. In case of hit, there is no overhead
cased by accessing data on disk. Hence, the I/O cost decreases when the hit rate increases.

Figure 13 shows the hit rate in accessing the PostgreSQL buffer cache during the extraction
process for two different configurations of the Kosarak dataset. With the default size of the buffer
 cache (i.e., 64 blocks) the hit rate is quite high even when the extraction is performed with low
support thresholds. When increasing the buffer cache size, the hit rate grows by about 1.0% in
both cases (see Figure 13), but the performance for the extraction process slightly decreases (e.g.,
when the buffer cache is 512 blocks the CPU time required for the extraction process increases
by about 4%). In this case, the decrease in I/O cost is counterbalanced by a decrease in the
memory space available for the extraction algorithm.

6 Related Work

Incremental frequent pattern mining is a relevant research area which finds application in many
real-life contexts. Incoming data may be (i) single transactions [9] or (ii) transaction blocks
[6, 8, 13, 14]. In the latter case, a set of transactions is added to the database at the same time.
Hence, the data model is called block evolution [14]. The I-Forest index structure addresses the
extraction of frequent itemsets from a sequence of incoming data blocks. Hence, it addresses this
last case.
Figure 13: Buffer cache and hit rate for Kosarak dataset

Works in [6, 8, 13, 14, 35] address incremental frequent itemset extraction under block evolution. The FUP [8] algorithm, based on the Apriori [5] approach, required a large number of database scans. It has been improved by BORDERS [6], based on the notion of border set (a set \( X \) is a border set if all its proper subsets are frequent sets but \( X \) itself is not). Border sets allow the efficient detection of new frequent sets, thus reducing the number of candidates to count and the number of database scans. This algorithm stores the set of frequent itemsets and updates them when new data are inserted. Hence, a large solution set has to be stored. Furthermore, updating the solution set may require to re-scan the whole dataset. To deal with smaller solution sets, [35] stores only the maximal itemsets while [10] maintains the set of closed itemsets. The BORDERS approach has been improved by the ECUT (Efficient Counting Using TID-lists) algorithm [13]. To count the support of itemsets, ECUT retrieves only the relevant portion of the dataset by means of the TID-lists of 1-itemsets. The ECUT\(^+\) algorithm, also proposed in [13], materializes TID-lists of itemsets of size greater than one. The number of itemsets may be very large. The problem of selecting the optimal set of itemsets to be materialized is NP-hard. With the heuristic approach proposed in ECUT\(^+\), only the TID-lists of 1-itemset and 2-itemsets are materialized, thus improving the performance with respect to ECUT at the expense of more disk space.

A different incremental technique is based on arbitrary insertions and deletions of single transactions. In this scenario, [9] proposed an extension of the FP-Tree [20], where each new transaction is inserted in the existing tree by means of a heuristic approach without exploring the whole tree. However, after several insertions the updated structure is not as compact and efficient as the corresponding FP-tree.

The I-Forest index provides a compact structure to store all the information potentially required by the mining process. Furthermore, it allows selective access to this information. Hence, differently from the above approaches, it provides a flexible framework in which different types of analysis (e.g., mining only a subset of interesting blocks or items) can be performed. Thus, it may be straightforwardly integrated into the framework proposed in [14].

A parallel effort has been devoted to the design and implementation of the computationally intensive knowledge extraction task over static databases [5, 18, 20, 29]. Since the required memory increases significantly, memory based approaches are not suitable for mining sequences of incoming data blocks. In this context, disk based approaches [7, 11, 23, 24, 31] are more
suitable for the itemset extraction task. They exploit clever data structures to summarize the (static) database on disk. Efficient itemset extraction takes place on these ad-hoc data structures.

[31] first proposed a clever data structure stored on disk to enhance the frequent itemset extraction process. In particular, it proposed B+tree based indices to access data stored by means of either a vertical or a horizontal data representation. The focus is on frequent itemset extraction by means of both Apriori [4] and Eclat [36] algorithms. However, their performance results are usually worse, or at best comparable, to flat file mining. In [23] an index structure based on a variant of the signature file is proposed to encode a dataset. The index is used to drive the generation of candidate frequent itemsets. However, to check the actual frequency of candidate itemsets, a further access to the database is needed.

More recently, [11] proposed a frequent itemset disk-based mining algorithm. In [11], the dataset is represented by means of an Inverted Matrix stored on disk, which is mined to extract frequent itemsets. The inverted matrix structure seems particularly suited for very sparse datasets, characterized by a significant number of unique items (i.e., items with unitary support). No previous solutions are integrated into a DBMS kernel.

One step further towards the tightly coupled integration of data mining algorithms into a relational database system has been addressed in [3]. [3] proposed user-defined functions (UDFs) in SQL statements to selectively push parts of the computations into the database system. The objective was to avoid one-at-a-time record retrieval from the database, by avoiding both copying and process context switching costs. The problem of expressing association rules mining queries in SQL has been explored in [19, 26, 30]. These approaches proposed an extension of SQL (e.g., DMQL [19], MINE RULE [26]) to allow the user to express requests in an intuitive way. However, performance is low because of the high complexity of SQL modules. All the techniques operate at the application level and do not affect the DBMS. This kind of high level integration has the advantage of being flexible and portable but not optimal in performance. To improve the performance of mining activities [7] proposes an index structure, called I-Tree, to tightly integrate frequent itemset mining in PostgreSQL open source DBMS. However, the I-Tree index cannot be incrementally updated. We exploit the I-Tree as a building block of our approach. The I-Forest index allows (i) time constrained extraction, (ii) incremental mining of evolving databases, and (iii) smooth data evolution.

7 Conclusions and Future Work

I-Forest is a new hierarchical index structure suitable for itemset extraction on evolving databases, whose content changes over time through insertions (or deletions) of transactional data blocks. It is characterized by (i) a compact lossless representation of a set of data blocks, (ii) an appropriate physical organization for efficient data retrieval during the mining process, and (iii) an adaptive structure that fits different data distributions. Furthermore, an algorithm to enhance itemset mining by exploiting the proposed index structure has been presented. Time and support constraints may be enforced to drive the extraction process. The proposed disk-based data structure has been implemented into the PostgreSQL open source DBMS, by exploiting its physical level access methods. Experimental results show that frequent itemset mining based on the I-Forest index is efficient in terms of both extraction time and memory usage. For low support thresholds, it outperforms the Prefix-Tree algorithm accessing static data on flat file. Furthermore, the I-Forest index allows to perform different kinds of analysis (e.g., mining periodical data, mining selected data blocks) on evolving databases.
As future work, the following issues may be addressed:

**Compact structures suitable for different data distribution.** Currently, the I-Tree structure is adopted to represent all data blocks of I-Forest. Since IF-blocks are independent of each other, each data block may be represented by means of a different compact structure suitable for its own data distribution. We plan to include appropriate data structures for sparser data distributions (e.g., Patricia-Trie [29] and Inverted Matrix [11]).

**Integration with a mining language.** The primitives for index creation and frequent itemset extraction may be integrated with a query language for specifying mining requests, such as DMQL [19] or MINE RULE [26]. Our low-level primitives can provide an efficient database implementation of the basic extraction statements of these languages.

**References**


Notes

1We also considered different block partitioning strategies, e.g., Dataset-75+25, where the first block contains 75% of the transactions and the second 25%. These experiments are not reported, since results are not significantly different.

2The amount of memory required by the extraction process is read from file /proc/PID/mem (memory held by the PID process).

List of Figures

1 I-forest structure ............................................................. 4
2 Data block at time t=1 ......................................................... 5
3 Data block at time t=2 ......................................................... 5
4 IF-block \( I_1 \) for block 1 ..................................................... 5
5 IF-block \( I_2 \) for block 2 ..................................................... 5
6 Frequent itemset extraction time ........................................ 11
7 Frequent itemset extraction time ........................................ 12
8 Frequent itemset extraction time: Kosarak ......................... 12
9 Effect of block partitioning on frequent itemset extraction time: Kosarak .......... 12
10 Effect of block partitioning on frequent itemset extraction time .......... 13
11 Itemset extraction time with both time and support constraints .......... 13
12 Memory usage for Kosarak dataset .................................... 14
13 Buffer cache and hit rate for Kosarak dataset ...................... 15