An Effective Technique for IMine Indexing using FP-Bonsai Tree

Peer Fatima  Dr. Mohammed Sathik
Research Scholar in Computer Science
Principal Manonmaniam Sundaranar University Sathakathullah Appa College
Tirunelveli, India Tirunelveli, India
pfatimabaasha@gmail.com  mmdsathick@gmail.com

Abstract---A set of programs that allows to store, modify, and obtain information from a database. There are numerous types of data base management systems, which might be a very little system that run on personal computers to enormous systems that run on mainframes. With the huge increase in the amount of information, it is very difficult to manage these data. Hence there is a need for an effective indexing technique. The advantage of using index lies in the fact that index makes search operation perform very fast. In this paper proposed the IMine index (a common and compressed structure which presents close integration of item set mining) by using FP-Bonsai Tree and I-Tree. Previous approach have used Prefix Hash Tree (PHT), but it exhibit reduced locality of reference, i.e., the data to be accessed is distributed apparently at random in memory. Also PHT cause access patterns to jump around, hence this causes long delays and there is a wasteful on memory, since many of array positions possibly will contain empty linked lists. These drawbacks are solved with the use of FP-Bonsai Tree.

Keywords--- Item set-Mine (IMine), FP-Bonsai Tree, Item set-Tree, Reading Cost, FP-growth Algorithm, Item Set Extraction

1. Introduction

Databases are exceptional data storage and retrieval tool, which ranges from unsophisticated tables of information to complex, complicated collections of multiple linked databases. A database is generated and named according to the essentials of its application, and stores data in tables consisting of a prearranged collection of rows and columns. Data may be physically entered into databases, but is more normally written dynamically into databases by web sites or by associated software applications. Companies frequently manage continuously growing databases containing data like client information, financial statistics and performance analyses and the ever increasing nature of such databases warrants techniques for streamlining the process of locating exact pieces of information within the organization. Indexing [1] permits for the quick retrieval of particular data by queries, and is consequently a powerful tool for databases containing hundreds or even millions of records.

The momentum to index data [10] occurs because of the requirement to find particular information and then to retrieve it as competently as possible. If the complete dataset are kept in main memory, then there is no need for indexing. Since this is not achievable, and hence disk access times are very slower than main memory access times, then there is a requirement for the art of indexing.

An index would be any structure which can be utilized to signifiy information that can be used to resourcefully evaluate a query. Information in these circumstances may be made up of text, images, audio, video and mixture of these that include domain-specific datatypes.

With the huge quantity of data such as that in databases, indexes make locating and retrieving the data quicker and more effective. Entries in an index specify three information’s regarding the items they refer to:

- What the item is (“employee details on particular name”)
- Where the item is (“record number 12156” or “page 165”)
- How the item is stored (“in a successive series of records” or “as text on a page”)

Most sets of data can be indexed [15] in several different ways. To provide the most useful and efficient access to data, it is often critical to choose the right style of indexing. This is because no indexing method is optimal for every application. Effective use of indexes [3] requires establishing a balance between an efficient number of indexes and too many. Because indexes consume some amount of space, using too many can counteract the intention by slowing down the database. Indexes [4] possibly will be needless for minor databases, where the queries can find data rapidly and effortlessly. But indexes are dynamic and commanding tools for optimizing the performance of large databases.

In this paper, proposed an approach to maintain data mining queries. The IMine index (Item set-Mine index) [5] is an innovative data structure that offers a compact and complete representation of transactional
data supporting well-organized item set extraction.

The structure of the IMine index is described by two structures: FP-Bonsai Tree and Item set-Tree [19].

In this paper, FP-Bonsai Tree can be developed even enhanced within the distinguished FP-growth algorithm. FP-Bonsai Tree is used along with the Item set-Tree to support IMine index [17]. FP-Bonsai Tree is constructed using the recursive projecting method of FP-growth and the ExAnte data-reduction is persistent all over the computation. The entire FP-trees constructed recursively throughout the FP-growth computation can be pruned comprehensively by using the ExAnte property, attaining a computation with a lesser number of smaller trees. These tiny FP-tree, are acquired by growing and pruning are recognized as FP-bonsai.

2. Literature Survey

Yin-Ling Cheung et al., [7] suggested mining frequent itemsets without support threshold: with and without item constraints. In traditional association rules mining, a minimum support threshold is considered to be available for mining frequent itemsets [12, 13]. But, fixing such a threshold is characteristically tough. This makes an additional practical difficulty; it is to mine N k-itemsets with the maximum supports for k equal to a certain kmax value. The final output is the N-most interesting itemsets. Normally, it is very simple for users to conclude N and kmax value. The author proposed two new approaches, namely LOOPBACK and BOMO. Experimental observation proves that this technique provides better result than the existing Itemset-Loop algorithm, and the output of BOMO can be an order of magnitude enhanced than the original FP-tree algorithm, still with the supposition of an optimally chosen support threshold. The author also proposed the mining of "N-most interesting k-itemsets with item constraints." This permits the user to denote different degrees of interestingness for dissimilar itemsets. Experimental observations show that this proposed Double FP-trees algorithm, is very effective in solving this problem which depends on BOMO.

A novel parallel frequent itemset mining algorithm was recommended by Chen Xiaoyun et al., [18]. Frequent itemset mining is an essential and important matter in data mining field and can be utilized in many data mining tasks. The majority of these mining tasks need multiple passes over the database and if the database size is huge, which is typically the case; scalable elevated performance solutions involving multiple processors are needed. In this paper, the authors proposed a new equivalent frequent itemset [6] mining approach which is called HPFP-Miner. The proposed technique depends on FP-Growth and establishes modest communication overheads by efficiently partitioning the list of frequent elements list over processors. The reports of experiment prove that HPFP-Miner has good scalability and performance.

Syeda-Mahmood et al., [2] suggested interval hash tree: an efficient index structure for searching object queries in large image databases. As image databases develop huge in size, index structures for quick navigation grown to be important. Especially, when the objective is to find object queries in image databases under changes in pose, occlusions and false data, conventional index structures used in database become inappropriate. The author proposed a novel index structure known as interval hash tree, for finding multi-area object queries in image databases. The effectiveness of the index structure is verified for query localization in a huge image database.

A fast algorithm for frequent itemset mining using FP-trees was proposed by G. Grahne et al., [8]. Well-organized algorithms for mining frequent itemsets [9] are vital for mining association rules, also for many additional data mining tasks. Techniques for extracting frequent itemsets have been employed by means of a prefix-tree structure called as FP-tree, implemented for storing compressed information regarding frequent itemsets. Many experimental observations have illustrated that these techniques execute very well. In this technique, the author proposed an innovative FP-array method that significantly decreases the necessity of traversing FP-trees, thus acquiring enhanced performance for FP-tree-based algorithms. The proposed approach works particularly well for sparse data sets. Additionally, the author proposed a novel technique for mining all, maximal, and closed frequent itemsets [16]. This approach uses the FP-tree data structure along with the FP-array method effectively and integrates several optimization techniques. Experimental result proves that this technique is the best for many cases. Although this approach takes much memory when the data sets are sparse but it is the fastest technique when the minimum support is low. Furthermore, this technique is the fastest techniques and uses less memory than previous techniques when the data sets are dense.

Improved paralleled algorithm for mining frequent item-set used in HRM was presented by XuePing Zhang et al., [11]. This approach established the technique of multi-thread processing and a Multi-Threaded Paralleled frequent item-set extraction Algorithm - MTPA was implemented depending on FP-tree algorithm. It has been implemented in an enterprise human resources management organization. Based on the experiments of paralleled mining by utilizing increasing multi-thread processing, it is confirmed that MTPA which on the circumstance of multi-core processors can enhance the efficiency of frequent item-set mining successfully.

3. Methodology

3.1. IMine Index Structure

The structure of the IMine index is described by two structures: FP-Bonsai Tree and Item set-Tree. The two structures offer two levels of indexing. The FP-
Bonsai Tree is a tiny FP-tree which permits reading preferred I-Tree portions all through the extraction task. For every item, it stores the physical locations of all item occurrences in the I-Tree. Thus, it supports resourcefully loading from the I-Tree the transactions in relation comprising the item. The Item set-Tree (I-Tree) is a prefix-tree which symbolizes relation using a concise and lossless compressed structure.

3.1.1. Prefix Hash Tree

The FP-growth algorithm stores the real transactions from the database in a trie structure (prefix tree), and moreover stores a header table holding all items with their support and the start of a linked list going throughout the transactions that contain that item. This data structure is represented by FP-tree (Frequent-Pattern tree) [14]. For instance, consider the transaction database in Fig-1(a) and a minimal support threshold of 4. Initially, all infrequent items are eliminated from the database, every transactions are reorganized in support descending order and introduced into the FP-tree, resulting in the tree in Fig-1(b).

Specified a transaction database $D$ and a minimal support threshold $\sigma$, the group of all frequent itemsets with the similar prefix are represented as $I \subseteq \mathcal{J}$ by $\mathcal{F}[I](D, \sigma)$. FP-growth recursively creates for all singleton items $\{i\} \in \mathcal{T}(C_{freq})$ the set $\mathcal{F}[\{i\}](D, \sigma)$ by generating the so called $i$ -projected database of $D$. This database, represented $D^i$, contains all transactions in $D$ containing $i$, from which all items which come before $i$, concerning the support descending order, are eliminated. This $i$ -projected database, which is again stored as an FP-tree, is recursively extracted by FP-growth.

The most important trick exploited in FP-growth is that it only needs to find all singleton frequent itemsets in the given database. Then, for all such item, it generates the equivalent projected database in which again, only the (local) singleton frequent itemsets have to be found. This procedure goes on until no more (local) items are present. The FP-tree structure promises that all this can be completed competently. In
this technique, FP-growth implicitly generates numerous databases, denoted by FPtrees. The most important quality is that all these datasets (trees) can be condensed (pruned) using the ExAnte technique. These pruned FP-tree are known as FP-bonsai.

With the intention of acquiring the complete algorithm that discovers all itemsets fulfilling the specified constraints, the FP pruning algorithm must be called before the first line of the FP-growth algorithm. The fact that the database is stored as an FP-tree is not exclusively mentioned. That is for the reason that this is in fact not essential, but the FP-tree is basically the most efficient data structure for these algorithms to use. The following example shows how the pruning can be efficiently applied on the FP-tree structure.

Once the FP-bonsai has been constructed (i.e. once the fix-point of α and μ pruning has been accomplished) the entire frequent itemsets fulfilling the particular constraints using FP-growth can be effectively mined. The recursive arrangement of the FPgrowth based algorithm plays a vital role in the construction of FP-Bonsai tree and the ExAnte property is extremely amalgamated with the frequent itemset computation: not only the preliminary tree is a pruned tree, but also all the other projected trees, constructed throughout the recursive growing stage will be much more smaller in number and in size.

### 3.1.2. I-Tree

An efficient method to neatly store transactional records is to make use of a prefix-tree. Trees and prefix-trees have been commonly used in data mining and datawarehousing indices. This proposed implementation of the I-Tree is depends on the FP-tree data structure [8], which is provides a compact and lossless representation of relation R. But, as the two index components are intended to be independent, alternative I-Tree data structures can be easily incorporated in the IMine index.

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

Figure 1 shows the FP-Bonsai tree and Figure 2 shows the Item set-Tree. Nodes in I-Tree paths are sorted by lessening support of the equivalent items. For the items with the similar support, nodes are arranged by item lexicographical order. In the I-Tree, the regular prefix of two transactions is represented by a single path. For example, consider transactions 3, 4, and 9 in the illustrated data set. These transactions, once arranged as illustrated 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.

Nodes in the I-Tree are connected by pointers which permit selectively loading from disk the index portion essential for the extraction process. Each node contains three pointers to nodes in the tree. Each pointer stores the physical location of the

---

**Figure-2: Item set-Tree**
corresponding node. An arbitrary node (e.g., [p:3] in the example I-Tree in Fig-2, involves the following links: a) Parent pointer (continuous edge linking node [p:3] to node [d:5]). b) First child pointer (dashed edge linking node [p:3] to node [g:2]). When a node contains more direct descendants, this pointer points to the first child node inserted in the I-Tree. c) Right brother pointer (dotted edge linking node [p:3] to node [n:2]). When a node contains 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 permit equally bottom-up and top-down tree traversal, therefore enabling item set extraction with different types of constraints.

**Table I: Example data set**

<table>
<thead>
<tr>
<th>TID</th>
<th>ItemID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>g, b, h, e, p, v, d</td>
</tr>
<tr>
<td>2</td>
<td>e, m, h, n, d, b</td>
</tr>
<tr>
<td>3</td>
<td>p, e, c, i, f, o, h</td>
</tr>
<tr>
<td>4</td>
<td>j, h, k, a, w, e</td>
</tr>
<tr>
<td>5</td>
<td>n, b, d, e, h</td>
</tr>
<tr>
<td>6</td>
<td>s, a, n, r, b, u, i</td>
</tr>
<tr>
<td>7</td>
<td>b, g, h, d, e, p,</td>
</tr>
<tr>
<td>8</td>
<td>a, i, b</td>
</tr>
<tr>
<td>9</td>
<td>f, i, e, p, c, h</td>
</tr>
<tr>
<td>10</td>
<td>t, h, a, e, b, r</td>
</tr>
<tr>
<td>11</td>
<td>A, r, e, b, h</td>
</tr>
<tr>
<td>12</td>
<td>z, b, i, a, n, r</td>
</tr>
<tr>
<td>13</td>
<td>b, e, d, p, h</td>
</tr>
</tbody>
</table>

3.2. IMine Data Access Methods

The IMine index structure is independent of the implemented item set extraction approach. Therefore, variety of the modern approaches may be utilized, once data has been loaded in memory. The in-memory demonstration appropriate for the preferred extraction algorithm is employed (e.g., FP-tree for FP-growth, array-based structure for LCM). Based on the enforced support and/or item constraints and on the preferred approach for item set extraction, a different portion of the IMine index can be used.

Since IMine is a covering index, the original database is not at all used. The IMine index permits selectively loading into memory only the index blocks used for the local search. Therefore, it supports a decrease of disk reads. Because only a small portion of the data is actually loaded in memory, more memory space is available for the extraction operation.

3.3. IMine Physical Organization

The physical organization of the IMine index is intended to reduce the cost of reading the data required for the existing extraction process. The Prefix Hash Tree permits a selective access to the I-Tree paths of interest. Hence, the I/O cost is mostly given by the number of disk blocks read to load the necessary I-Tree paths.

When visiting the I-Tree, nodes are examined from table TL_TreeNode with the help of their correct physical location. But, fetching a particular record needs to load the complete disk block where the record is stored. Alternatively, once the block is in the DBMS buffer cache, reading the further nodes in the block does not involve additional I/O cost. So, to decrease the I/O cost, correlated index parts, i.e., parts that are accessed collectively throughout the extraction task must be clustered into the same disk block. The I-Tree physical organization is based on the following correlation types.

- **Intratransaction correlation**: An Extraction algorithm believes together items taking place in the same transaction. Items appearing in a transaction are thus basically correlated. To diminish the number of read blocks, each I-Tree path is supposed to be partitioned in a limited number of blocks.
- **Intertransaction correlation**: Transactions with some items in general will be accessed collectively when item sets including the familiar items are extracted. Hence, they should be stored in the similar blocks. In the I-Tree, the familiar prefix of different transactions is denoted by a single path. Additionally to enhance transaction clustering, subpaths with a specified percentage of items in general should be stored in the similar disk block.

4. Experimental Results

**Computation Time:**

The computational time of the proposed approach using the FP-Bonsai tree is compared with the existing techniques. The FP-Bonsai tree takes very less computation time when compared with the existing approaches. Fig-3 shows the graphical representation of the comparison of the computational time of the proposed approach with the existing approach.

**Size of the Index:**

The size of an index plays a major role in the performance of the IMine index. It is very difficult to search a required node when the size of an index is huge. Since this approach uses FP-Bonsai tree in the IMine index, the size of index structure is significantly reduced. Table II shows the comparison of the size of
the standard datasets and the size of same datasets after using FP-Bonsai tree in IMine.

![Graph showing comparison of computational time](image)

**Figure-3:** Comparison of the Computational Time

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Standard Size (KB)</th>
<th>Index Size after using FP-Bonsai tree in IMine (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td>25527</td>
<td>18256</td>
</tr>
<tr>
<td>PUMSB</td>
<td>35829</td>
<td>23152</td>
</tr>
<tr>
<td>KOSARAK</td>
<td>85435</td>
<td>59635</td>
</tr>
<tr>
<td>T101200P20D2M</td>
<td>544326</td>
<td>376525</td>
</tr>
<tr>
<td>T151100P20C1D5M</td>
<td>1476523</td>
<td>958526</td>
</tr>
<tr>
<td>T201100P15C1D7M</td>
<td>2075478</td>
<td>1552466</td>
</tr>
</tbody>
</table>

This performance is primarily because of the optimized storage of data in a small number of blocks, which allows selective data loading in memory. Hence, I/O costs are reduced and considerably more memory space is accessible for the extraction task.

**5. Conclusion**

Due to the world wide increase in the available data, it is very difficult for obtaining the related data with better accuracy. Hence there is a need for an effective indexing technique. It is very easy to mine with the help of indexing. Therefore, IMine index is developed by using FP-Bonsai tree and Item set-Tree. The IMine index makes the process of storage, retrieval and accessing of the information very easier. In this paper, FP-Bonsai tree is used in the IMine index structure, which is a FP-growth tree and computation can be pruned comprehensively by using the ExAante property. The computational time of the proposed approach is very less when compared with the previous approaches. The size of an index is considerably reduced with the use of FP-Bonsai tree.

**References**


