

# Mining Frequent Item Sets by Opportunistic Projection<sup>#</sup>

Junqiang Liu \* Yunhe Pan

Institute of Artificial Intelligence  
Zhejiang University

liujunq@mail.hz.zj.cn

panyh@sun.zju.edu.cn

Ke Wang

School of Computing Science  
Simon Fraser University

wangk@cs.sfu.ca

Jiawei Han

Department of Computer Science  
UIUC

hanj@cs.uiuc.edu

## ABSTRACT

In this paper, we present a novel algorithm OpportuneProject for mining complete set of frequent item sets by projecting databases to grow a frequent item set tree. Our algorithm is fundamentally different from those proposed in the past in that it opportunistically chooses between two different structures, array-based or tree-based, to represent projected transaction subsets, and heuristically decides to build unfiltered pseudo projection or to make a filtered copy according to features of the subsets. More importantly, we propose novel methods to build tree-based pseudo projections and array-based unfiltered projections for projected transaction subsets, which makes our algorithm both CPU time efficient and memory saving. Basically, the algorithm grows the frequent item set tree by depth first search, whereas breadth first search is used to build the upper portion of the tree if necessary. We test our algorithm versus several other algorithms on real world datasets, such as BMS-POS, and on IBM artificial datasets. The empirical results show that our algorithm is not only the most efficient on both sparse and dense databases at all levels of support threshold, but also highly scalable to very large databases.

## Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications – Data Mining.

## General Terms

Algorithms

## Keywords

Association Rules, Frequent Patterns

## 1. INTRODUCTION

Mining frequent item sets is a key step in many data mining problems, such as association rule mining, sequential pattern mining, classification, and so on. Since the pioneering work in [3], the problem of efficiently generating frequent item sets has been an active research topic.

Let  $I = \{i_1, i_2, \dots, i_m\}$  be a set of literals, called items. Let database  $D$  be a set of transactions, where each transaction  $T$  is a set of

items such that  $T \subseteq I$ . Each transaction is associated with a unique identifier, called TID. Let  $X$  be a set of items. A transaction  $T$  is said to contain  $X$  if and only if  $X \subseteq T$ . The support of a set of items  $X$  is the number or the percentage of transactions in the database that contain  $X$ .  $X$  is frequent if the support of  $X$  is no less than a user defined support threshold. We are interested in finding the complete set of frequent item sets.

Frequent item sets can be organized as a tree that is not necessarily materialized. Mining process can be regarded as a process of frequent item set tree growth accompanied by a process of projecting transaction subsets. In the light of this framework, all algorithms either grow the frequent item set tree by a breadth first approach or by a depth first approach.

Apriori [4] is a prominent breadth first algorithm, followed by many variants that improve Apriori by reducing the number of candidates further [11], the number of transactions to be scanned [4,8,11], or the number of database scans [7,13,14]. TreeProjection [1] is the latest breadth first algorithm. However, breadth first algorithms are inefficient for dense datasets that contain long patterns. Recently, the merits of a depth first approach have been recognized [6], a few algorithms are proposed [2,6,9,12]. However, algorithms proposed so far do not fully exploit the strength of depth first search and do not scale to large sparse databases yet. [2,5,6,10] propose algorithms that output only maximal frequent patterns by pruning the frequent item set tree based on superset frequency. However, maximal frequent patterns have limitations in generation of association rules.

The representation of projected transaction subsets can be array-based [12], tree-based [9], vertical bitmap [6], or horizontal bitstring [2], which is the key factor in the efficiency of projection operation and counting operation. None is good for all situations. Actually, the maximized efficiency and scalability are achieved by balancing the tradeoffs between different representation forms and associated projecting methods and counting methods in different situations.

In this paper, we present a novel algorithm, OpportuneProject, for mining complete set of frequent item sets, which is efficient on both sparse and dense databases at all levels of support threshold, and scalable to very large databases. Our contributions are as follows. First, we present novel pseudo projection methods for tree-based representations in the depth first search, which greatly improves the efficiency of counting and projecting operations in dense transaction subsets. Second, we propose an array-based data structure that is the most space efficient and the simplest for sparse datasets. Third, we define heuristics that adapts the algorithm to the features of the projected transaction subsets by integrating array-based and tree-based representations, and employing different projecting and counting methods opportunistically. Finally, we use a hybrid approach to deal with

\* Also an associate professor at Hangzhou University of Commerce.

# This work is supported in part by the NSF of Zhejiang, China, and the NSERC of Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGKDD '02, July 23-26, 2002, Edmonton, Alberta, Canada.

Copyright 2002 ACM 1-58113-567-X/02/0007...\$5.00.

very large databases, i.e., to grow the upper portion of the frequent item set tree by breadth first search and grow the lower portion by guided depth first search.

## 1.1 Related works

[1] presents a method, TreeProjection, which represents frequent patterns as nodes of a lexicographic tree and uses the hierarchical structure of the lexicographic tree to successively project transactions and uses matrix counting on the reduced set of transactions for finding frequent patterns. The algorithm looks only at the subset of transactions, which can possibly contain the pattern by traversing the lexicographic tree in a top down fashion. This significantly improves the performance of counting the number of transactions containing a frequent pattern. TreeProjection is primarily based on pure breadth first strategy. It encountered the same problems as Apriori, such as high cost for pattern matching incurred by projecting on the fly, huge frequent item set tree, and too many database scans.

[9] presents a well known depth-first algorithm, FP-growth, which is reported to be faster than TreeProjection and Apriori. It first builds up a compressed data structure, FP-tree, to hold the entire database in memory and then recursively builds conditional FP-trees to mine frequent patterns. It has performance gains since it avoids the combinatory problem inherent to candidate generate-and-test approach. However, the number of conditional FP-trees is in the same order of magnitude as number of frequent item sets. The algorithm is not scalable to sparse and very large databases.

[12] proposes a memory-based hyper structure, H-struct, to store the sparse databases in main memory, and develops an H-struct based pattern-growth algorithm, H-Mine. H-Mine invokes FP-Growth to mine dense databases, hence, suffers the inefficiency caused by recursive creations of conditional FP-tree. H-Mine uses partition-based method to deal with very large databases. Because the number of local frequent patterns in all partitioned databases can be huge, H-Mine still encounters great difficulties for very large databases.

DepthProject [2] and MAFIA [6] are two new algorithms that find maximal frequent item sets by depth first search. DepthProject employs a selective projection and uses the horizontal bitstring representation for projected transaction subsets, whereas MAFIA uses the vertical bitmap representation with a bitmap compression schema. Both improve the efficiency of counting over the naïve counting method by a factor of 8. However, they are less efficient than the array-based representation when the average number of items in transactions is sufficiently less than the total number of items, which is usually the case for sparse and large databases. On the other hand, the compression ratio of the tree-based representation is significant for dense databases in that a node represents a relatively large number of items. Therefore, item counting in the tree-based representation is more efficient than, at least comparable to, in the bitstring and bitmap representations for dense databases. Moreover, the pseudo projection method is more efficient than the selective projection in DepthProject and the compression schema in MAFIA.

The organization of the paper is as follows. Section 2 defines the frequent item set tree and discusses projection strategies. Section 3 begins with introducing an array-based representation for sparse projected transaction sets and the corresponding projection method. Then, novel methods for pseudo projection of tree-based

representation are developed. Based on observations and heuristics, the algorithm OpportuneProject is presented, which maximizes efficiency and scalability on databases of all features. In Section 4, the algorithm is evaluated experimentally. Section 5 concludes this paper.

## 2. PROBLEM DESCRIPTIONS

Frequent item sets can be represented by a tree, namely frequent item set tree, abbreviated as FIST, which is not necessarily materialized. In order to avoid repetitiveness, we impose an ordering on the items.

FIST is an ordered tree, where each node is labeled by an item, and associated with a weight. The ordering of items labeling the nodes along any path (top down) and the ordering of items labeling children of any node (left to right) follow the imposed ordering. Each frequent item set is represented by one and only one path starting from the root, and the weight of the ending node is the support of the item set. The null root corresponds to the empty item set. For example, the path  $(.)-(c,4)-(f,3)-(m,3)$  in Figure 1 represents the item set  $\{c, f, m\}$  with support of 3. The weights associated with nodes need not be actually implemented.

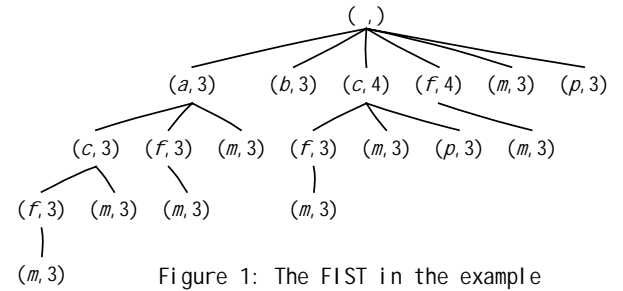


Figure 1: The FIST in the example

Mining frequent item sets can be regarded as a process of FIST construction, which is facilitated by successively projecting the transactions in a top down fashion. Figure 2 illustrates the basic idea by an example (the support threshold is set to 3).

Each node has its own projected transaction set (abbreviated as PTS). PTS consists of transactions that support the item set represented by the path starting from the root to the node. PTS of the null root is the original database. PTS of any node other than the null root is obtained by projecting transactions in PTS of its parent node, according to the a priori property. For example, the item  $a$  in original database in Figure 2 has a support of 3 that comes from transaction 01, 02, and 05. Hence, PTS of the child node  $(a,3)$  of the null root consists of these three transactions.

One PTS is filtered if each transaction in the PTS only maintains items that contribute to the further construction of descendants. In other words, filtered PTS of a node only contains items that label the sibling of its parent node. Otherwise, the PTS is unfiltered. Apparently, items in filtered PTS are local frequent in its parent PTS. In Figure 2, the PTS of the null root is unfiltered, and all other PTSs are filtered.

Basically, FIST can be created either by breadth first search or by depth first search. In breadth first search, all nodes at level- $k$  are created before nodes at level- $(k+1)$ . The PTS is a small subset of the original database for each node. However, the total space occupied by the PTSs over all nodes at the given level is much larger than the original database size. Thus, algorithms follow this

strategy usually do not maintain PTSs in the memory nor on the disk, they create PTSs on the fly. In other words, they read a transaction from the database into the memory, recursively projects the transaction from the null root down to the given level. This is a CPU-bound pattern matching task. Moreover, breadth first algorithms have to maintain the entire FIST in the memory. For dense database or for low support threshold, the huge size of FIST will exceed the capacity of the memory. The CPU-bound pattern matching and memory-bound FIST size are inherent to breadth first search strategy, even the original database can be loaded into the main memory. The advantage of breadth first search is that it is scalable to very large size of original database.

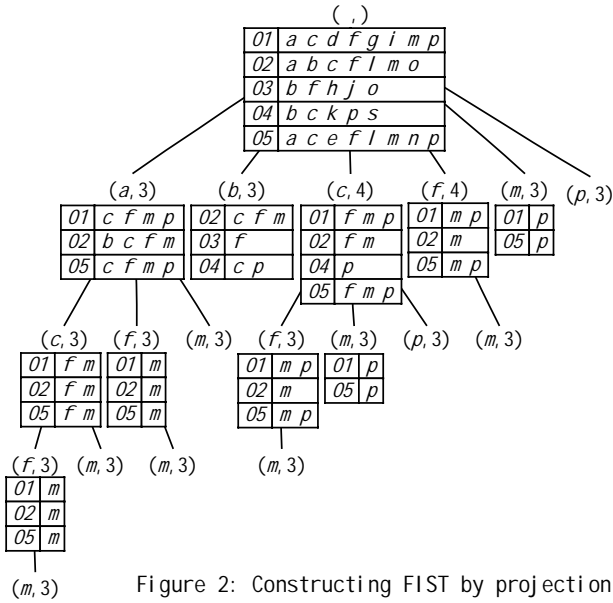


Figure 2: Constructing FIST by projection

In depth first search, PTSs are maintained for all nodes on the path starting from the root to the node that is currently being explored. Depth first search has the advantage that it is not necessary to re-create PTSs. This avoids the CPU-bound pattern matching inherent to breadth first search. Moreover, only the branch that is currently being explored needs to be maintained in the memory. This overcomes the limitation on the size of FIST. Depth first search is especially efficient for dense database and for low support threshold. Depth first search is usually memory based, that is PTSs are maintained in the memory. Hence, depth first search is not scalable to very large databases. Generally speaking, depth first search is more efficient, and breadth first search is more scalable.

### 3. MINING FREQUENT ITEM SETS BY OPPORTUNISTIC PROJECTION

To achieve maximized efficiency and scalability, the algorithm must adapt the construction strategy of FIST, the representation of PTS, and the methods of item counting in and projection creating of PTSs to the features of PTSs. In this section, an array-based PTS representation and projecting method is discussed firstly, to find complete set of frequent items by depth first search in sparse and large databases. Secondly, novel methods for projecting tree-based PTS representation are detailed, which is highly efficient for dense databases. Thirdly, observations and heuristics are

discussed. This section culminates in the presentation of the algorithm OpportuneProject that employs a hybrid approach.

#### 3.1 Mining sparse data by projecting array-based PTS

We use an array-based simple structure TVLA (threaded varied length arrays) to represent PTSs. TVLA consists of three parts: a local frequent item list (FIL), linked queues (LQ), and arrays.

Each local frequent item has an entry in the frequent item list (FIL), with three fields: an item-id, a support count, and a pointer. Entries in FIL are ordered by the imposed ordering. Each transaction is stored in an array and items in the array are sorted by the same ordering as FIL. Transactions with the same heading item are threaded together by a linked queue (LQ) which is attached to the entry with the same item in FIL. Apparently, the heading item need not be stored in the array, and the LQ points to the item next to the heading item in the transaction. For example, Figure 3(a) shows the filtered TVLA for the PTS of the null root of the FIST in Figure 2.

It is simple to get a child node's PTS from its parent node's PTS in the TVLA form. First, the transaction arrays that support a node's first child are already threaded by the LQ attached to the first entry of FIL. For example, the LQ(a) in the Figure 3(a) threads transactions 01, 02, and 05 that support the first child node (a,3) of the null root. The LQ(b) only threads part of transactions that support the child node (b,3) at that time.

Second, by shifting transactions threaded in the LQ that are currently explored to subsequent LQs, we can get PTSs that support the second child, and so on. A transaction is shifted by threading it into a proper LQ according to the item next to the heading item. For example, transaction 01 shifts from LQ(a) in Figure 3(a) to LQ(c), transaction 02 and 05 shift to LQ(b) and LQ(c) respectively. Then, we get the Figure 3(b) where LQ(b) threads all transactions that support the second child (b,3) of the null root. Following the same procedure step by step, we get PTSs that support the remaining children of the null root as shown by Figure 3(c) through 3(f).

A child TVLA has a local FIL, local LQs, and can share transaction arrays with its parent TVLA. In this case, it is unfiltered. A filtered TVLA has its local copy of transactions that trims out items irrelevant to further projection. For example, the TVLA in Figure 4(a) is the unfiltered child TVLA that represents PTS of the child node (a,3) of the null root in Figure 2. Figure 4(b) is the filtered child TVLA. TVLA is space efficient in representing PTSs for sparse database. Projecting and counting in TVLA is also very efficient since sparse databases shrink quickly.

#### 3.2 Intelligent projecting tree-based PTS depth first

In this subsection, we discuss a tree-based representation for dense PTS, which is inspired by FP-Growth. But, novel methods we proposed to project tree-based PTS is totally different from recursive creation of conditional FP-trees in FP-Growth.

##### 3.2.1 Representing dense PTS by TTF

A threaded transaction forest, TTF, is a representation of PTS, which consists of two parts: an item list (IL), and a forest.

Each local item in PTS has an entry in the IL, with three fields: an item-id, a support count, and a pointer. Entries in IL are ordered by the imposed ordering. Each transaction in the PTS is represented by one and only one path in the forest. Each node in the forest is labeled by (i, w) where i is an item and w is a count that is the number of transactions represented by the path starting from a root ending at the node. Items labeling nodes along any path are sorted by the same ordering as IL. All nodes labeled by the same item are threaded by the entry in IL with the same item. TTF is filtered if only local frequent items appear in TTF, otherwise unfiltered.

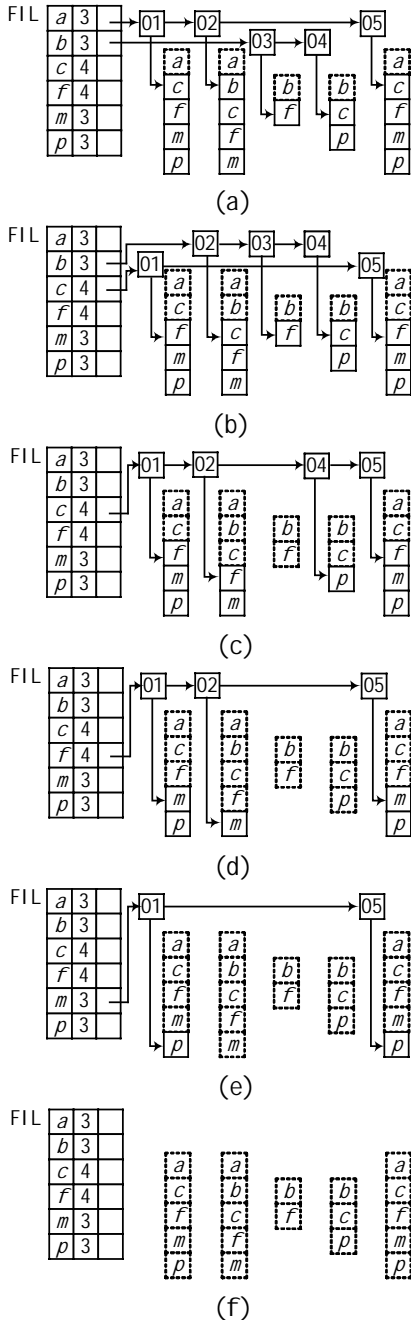


Figure 3. Representing PTS by TVLA

For example, the filtered TTF representation for the PTS of the null root in Figure 2 is shown in Figure 5(a), where the path (a,3)-(c,2)-(f,2)-(m,2)-(p,2) represents transaction 01 and 05, (a,3)-(b,1)-(c,1)-(f,1)-(m,1) represents transaction 02, and so on.

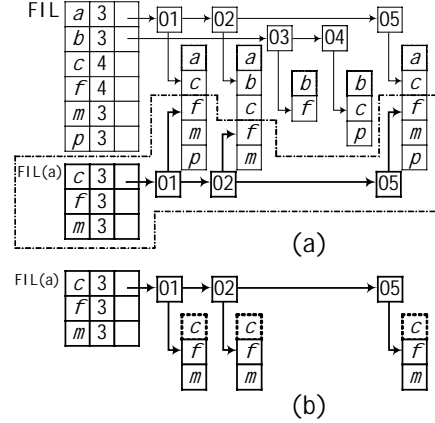


Figure 4. Unfiltered vs. filtered TVLA

### 3.2.2 Bottom up pseudo projection of TTF

From the TTF representation of the PTS of a parent node in FIST, we can project its children's TTFs either in a bottom up way or in a top down way.

In the bottom up way, the child TTF is derived from sub forest rooted at nodes that are threaded together in its parent TTF. For example, the sub forest rooted at the node (c,2), (c,1) and (c,1) that are threaded by the third entry of IL in Figure 5(d) compresses all transactions of the PTS of the third child (c,4) of the null root in Figure 2. By traversing the sub forest, we can count support for items in the child TTF, and re-thread nodes of the child TTF, clearly delimitate the PTS. For example, in Figure 5(d), all nodes and entries of IL in bold face, excluding IL entry and nodes of item c, are of the child TTF. The entries of item f, m, and p in the IL accumulate these items' support in the child TTF.

We call such a child TTF a pseudo projection of its parent TTF in that the child TTF, including its IL and forest, need not be materialized separately. The child TTF shares the same memory space with its parent TTF. This is a recursive procedure. For example, we can project the pseudo TTF in Figure 5(d) further and get the pseudo TTF in Figure 6(c) that represents the PTS of the first child node (f,3) of node (c,4) in Figure 2.

It is important to notice that the children of a node in FIST is arranged in accordance with the imposed ordering of items from left to right, whereas the children's pseudo TTFs are explored in a reverse ordering. This carefully chosen order of exploration is named as bottom up. For example, for the null root in Figure 2, we explore its last child (p,3) first, and its first child (a,3) at last, as shown in Figure 5(a) through (f). For the same reason, the children's TTFs of node (c,4) is projected in the order of Figure 6(a) through (c).

### 3.2.3 Top down pseudo projection of TTF

In the top down way, the pseudo TTF of a child PTS consists of sub forest whose leaves are threaded together in its parent TTF. For example, sub forest together with the corresponding entries of IL in bold face in Figure 7(a) through (f) are pseudo TTFs of

children of the null root projected by top down exploration of the parent TTF. The key points of top down exploration of pseudo TTF are as follows.

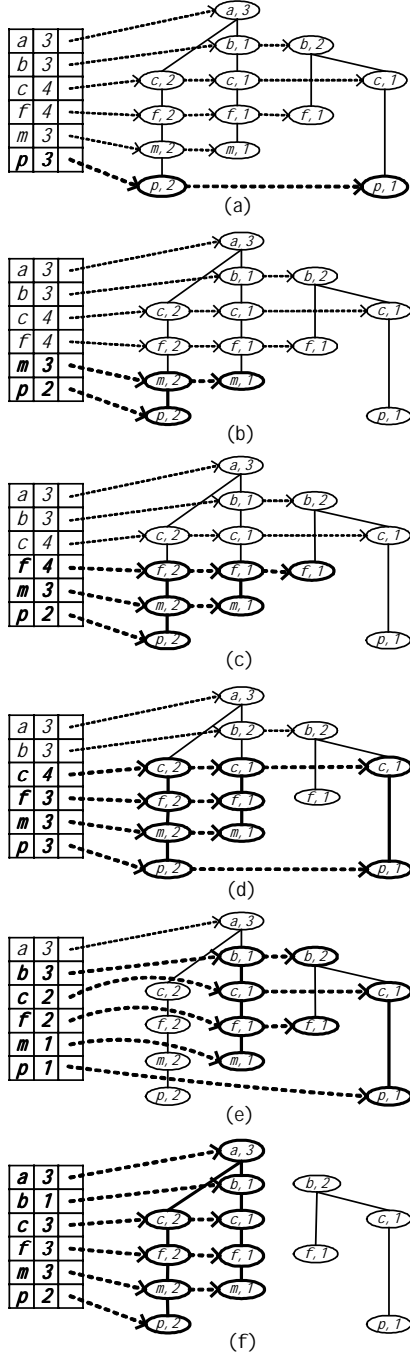


Figure 5. Bottom up pseudo projecting TTF

First, any pseudo TTF consists of a sub forest of its parent TTF and leaves of the sub forest are label by the same item and threaded by the entry of IL with the same item.

Second, by traversing the sub forest, we can delimitate the PTS by re-threading nodes in the sub forest, count the support of each

item in the sub forest by re-calculating the count of each node according to the leaves' support. Let us call the procedure as DelimitateSubForest. For example, in Figure 7(d), the sub forest whose leaves, (f,2), (f,1) and (f,1) are threaded by the entry of item f, compresses transactions that support item f. By traversing this sub forest, we get local support of item a, b, c of 3, 2, 3 respectively, and the count of second node label by b is changed from 2 to 1. Therefore, the sub forest of the pseudo TTF consists of three paths, (a,3)-(c,2), (a,3)-(b,1)-(c,1), and (b,1). The IL is {(a,3,ptr), (b,2,ptr), (c,3,ptr)}.

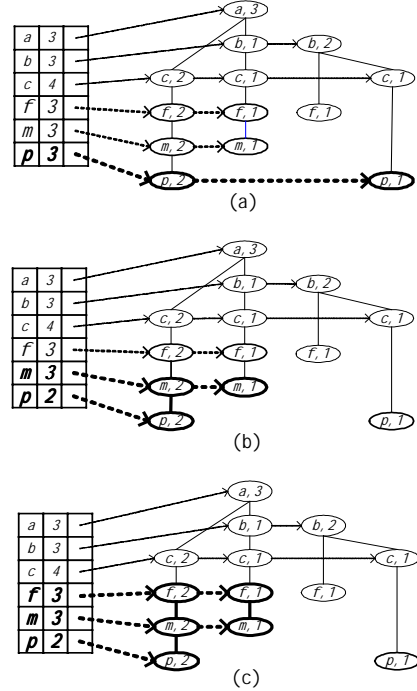


Figure 6. Children of the TTF in Figure 5(d)

Third, since DelimitateSubForest will change the threads of entries that are before the current entry in the IL, and change the count of internal nodes in the sub forest, it is very important to explore sub forests threaded by entries of IL in a particular TTF in the top down order. For example, the pseudo TTF shown in Figure 7(d) has three entries for the local item a, b, and c. We explore the sub forest threaded by the entry of item a first, and then the sub forest threaded by the entry of item c. Therefore, we get the pseudo child TTFs as in Figure 8(a), (b).

Fourth, the order of frequent item sets found by top down pseudo projection of TTFs is different from by bottom up pseudo projection as shown in Figure 9.

Our novel methods of pseudo projection avoid recursively building projected transaction set, which is in the same number as frequent item sets. These methods are not only space efficient in that no additional space is needed for any child TTF, but the counting and projecting operation is also highly CPU-efficient. Empirical study shows that algorithms based on these methods are much more efficient than Apriori and FP-Growth.

### 3.3 Observations and Heuristics

Real databases are skew, which cannot be simply classified as purely sparse or purely dense. Some PTSs of real database are

dense. Some are sparse. Real databases are of all sizes. We are going to propose a hybrid approach that maximizes efficiency and scalability for mining real databases, based on following observations and heuristics.

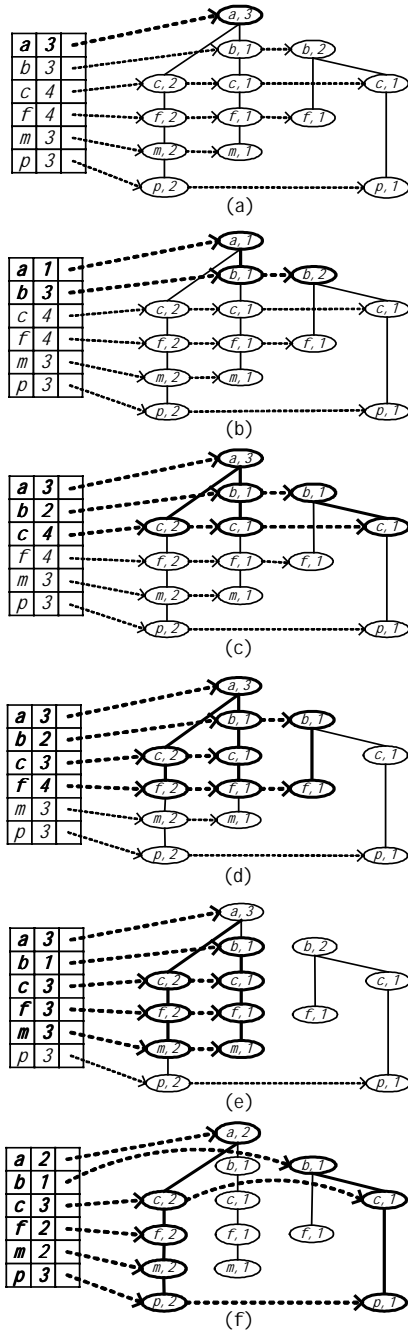


Figure 7. Top down pseudo projecting TTF

Observation 1: For very large databases, it is unrealistic to assume that the original database can fit in memory. The number of transactions that support item sets of length  $k$  decreases sharply when  $k$  is greater than 2. It is reasonable to assume that the upper portion of a FIST can fit in memory. Therefore, we can employ breadth first search to reduce transactions although most

algorithms use database partition method. It is noticed that breadth first search incurs heavy CPU cost since PTSs are projected on the fly by pattern matching transactions against paths on FIST.

Heuristic 1: Grow the FIST breadth first for very large databases. Whenever the reduced transaction set that support all nodes at level  $k$  can be represented by a memory based structure, either TVLA or TTF, grow the lower portion under level  $k$  depth first.

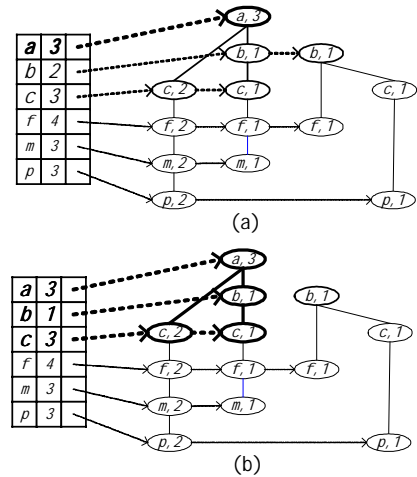


Figure 8. Children of the TTF in Figure 7(d)

Observation 2: Transactions in PTSs of nodes at high levels on FIST are usually diversified and randomly distributed. They have less chance to share common prefix with each other. TTF does not compress transactions well. Since TTF needs much more additional storage overhead than TVLA, TTF is usually space expensive relative to TVLA at high levels on FIST. On the other hand, it has been noted that the larger the relative support threshold, the larger the compression ratio of TTF. At lower levels or denser branches on FIST, there are fewer local frequent items in PTSs and the relative support is larger, TTF compresses well.

Heuristic 2: Represent PTSs at high levels on FIST by TVLA, unless the estimated compression ratio of TTF is sufficiently high.

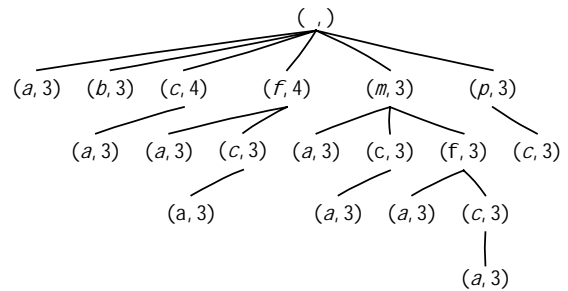


Figure 9. Build FIST by top down pseudo projecting TTF

Observation 3: PTSs shrink very quickly at high levels or sparse branches on FIST where PTSs are usually in the form of TVLA. The filtered TVLA is much more efficient than unfiltered TVLA for further counting and projecting operations. On the other hand, PTSs at lower levels or dense branches on FIST shrink slowly where PTSs are represented by TTF. The creation of filtered TTF involves expensive pattern matching operations.

Heuristic 3: When projecting a parent TVLA, make a filtered copy for the child TVLA as long as there is free memory. When projecting a parent TTF, delimitate the pseudo child TTF first and then make a filtered copy if it shrinks substantially sharp.

### 3.3.1 Estimate the size of TVLA and TTF

For a given PTS, let the number of frequent items be  $f$ , the number of transactions be  $t$ , and total number of occurrences of frequent items be  $o$ .

The exact size of TVLA is  $3*f + 2*t + (o-t)$ , where  $3*f$  is the size of FIL,  $2*t$  is the size of LQs, and  $(o-t)$  is the size of arrays.

The size of TTF is  $3*f + 6*n$ , where  $n$  is number of nodes of TTF. However, the exact number of nodes of TTF is unknown before its creation. The following formula gives the worst estimate of nodes of TTF, where  $u$  and  $l$  are the maximal and minimal length of filtered transactions.

$$n = \sum_{i=1}^u C_f^i - \sum_{i=1}^{l-1} C_{f-i-1}^{l-i-1} (2^i - 1) \leq 2^{f-1}$$

In our algorithm we estimate  $u$  and  $l$  based on the average transaction length. Numerous experiments show this estimation is always larger than the actual size. In other words, this is a pessimistic measure. The compression ratio of TTF is  $r = o/n$ . If  $r$  is less than  $6-(t/n)$ , the size of TTF is greater than TVLA, which is the case for sparse databases.

## 3.4 Algorithm OpportuneProject

Now we present the algorithm OpportuneProject, abbreviated as OP, which integrates depth first and breadth first strategy, array-based and tree-based representation, pseudo unfiltered projection and filtered projection, as listed in Figure 10.

### 3.4.1 The Breadth First Search

We create the upper portion of FIST in three steps. First, CreateCountingVector( $v$ ). We attach counting vectors to all nodes at the current level  $k$  to accumulate local supports for items in the PTS of each node. The counting vector has an element for the item of each sibling node that is after the node attached according to the imposed ordering. For example, possible items local to the PTS of the node (a,3) in Figure 1 are b, c, f, m, and p, which are the items of siblings that follow the node (a,3). Therefore, a length 5 counting vector is attached to accumulate the supports for item b, c, f, m, and p.

Second, ProjectAndCount( $t, D'$ ). We project the transaction  $t$  along the path from the root to nodes at the current level  $k$  and accumulate counting vectors. If a transaction can be projected to a level  $k$  node and contribute to its counting vector, it may also be projected to level  $k+1$ , therefore record it in  $D'$ . Otherwise it can be removed from further consideration. This results in the reduction of the number of transactions level by level.

Third, GenerateChildren( $v$ ). We create children for each node at the current level  $k$  for its local frequent items whose element in the counting vector has a value over the support threshold. If the node  $v$  has no child, it is removed at that time, and its parent will be deleted also if  $v$  is the only child of its parent, and so on.

The BreadthFirst is a recursive procedure. We use the available free memory as parameter to control breadth first search process.

```

OpportuneProject(Database: D)
begin
  create a null root for frequent item set tree T;
  D' = BreadthFirst(T, D);
  v = the null root of T;
  GuidedDepthFirst(v, D');
end
BreadthFirst(FIST: T, CurrentLevel: L, Database: D)
begin
  for each node v at level L of T do
    CreateCountingVector(v);
  D' = { };
  for each transaction t in D do
    ProjectAndCount(t, D');
  for each node v at level L of T do
    GenerateChildren(v);
  if D' cannot be represented by TVLA and TTF
  then BreadthFirst(T, L+1, D');
  else return(D');
end
GuidedDepthFirst(CurrentFISTNode: p, PTS: D)
begin
  ILp = TraverseAndCount(D, p);
  Dp = Represent(D, p);
  For each frequent entry e in ILp by particular ordering do
  begin
    c = GetChild(p, e);
    GuidedDepthFirst(c, Dp);
  end
end

```

Figure 10. Algorithm OpportuneProject

### 3.4.2 The Guided Depth First Search

Suppose the BreadthFirst procedure stops at level  $k$ . Then, only paths with length of  $k$  are maintained on the FIST whose lower portion will be generated by GuidedDepthFirst as follows.

First, TraverseAndCount( $D, p$ ) scans all transactions in  $D$  that support  $p$ , namely  $Dp$ , and get  $ILp$  which either be local frequent item list created at that time if  $D$  is on the disk or in the form of TVLA, or be represented in parent  $IL$  if  $D$  is in the form of TTF.

Second, Represent( $D, p$ ). If  $D$  is on the disk or in the form of TVLA, create a TTF for  $Dp$  if the density of  $Dp$  is estimated to be greater than a given value, otherwise create a filtered TVLA. If  $D$  is in the form of TTF, represent  $Dp$  by a pseudo TTF, and make a filtered copy if necessary.

Third, GetChild( $p, e$ ), for node  $p$ , either retrieve a child  $c$  that is labeled by the same item as that of  $e$  if the child is already created by BreadthFirst procedure, otherwise create the child at that time.

The GuidedDepthFirst procedure is more efficient than unguided one in that it avoids re-creating paths that end at the upper portion created by the BreadthFirst procedure.

## 4. PERFORMANCE EVALUATIONS

To evaluate the efficiency and effectiveness of our algorithm OpportuneProject, we have done extensive experiments on various kinds of datasets with different features by comparing with Apriori [16], FP-Growth [9], and H-Mine [12] on a 800MHz Pentium IV PC with 512MB main memory and 20GB hard drive, running on Microsoft Windows 2000 Server.

## 4.1 Datasets and features

We now describe the datasets used in our experiments. The basic features of the datasets are listed in Table 1.

BMS-POS, BMS-WebView-1 and BMS-WebView-2 are real world datasets and categorized as sparse datasets [15]. BMS-POS dataset contains several years worth of point-of-sale data from a large electronics retailer. The transaction in this dataset is a customer's purchase transaction consisting of all the product categories purchased at one time. BMS-POS has 122,449 frequent patterns at the support threshold of 0.1%, and 984,531 at 0.04%. BMS-WebView-1 and BMS-WebView-2 datasets contain several months worth of click stream data from two e-commerce web sites. BMS-WebView-1 has 3,991 frequent patterns at the support threshold of 0.1%, and 1,177,607 at 0.058%. The BMS-WebView-2 has 23,294 frequent patterns at support threshold of 0.1%, and 1,316,614 at 0.02%.

Connect4 is from UCI Machine Learning Repository [18]. Each transaction in Connect4 contains legal 8-ply positions in the game of connect-4 where neither player has won yet and the next move is not forced. It is a very dense dataset in that the number of frequent patterns grows from 27,127 to 4,129,839 and 88,316,367 when support threshold reduces from 90% to 70% and 50%.

IBM Artificial datasets, T25I20N20kL5k with D100k~D15m are generated using a transaction data generator [4] obtained from IBM Almaden [17]. T25I20N20.L5k can be regarded as something between the sparse and the dense. For example for D100k, the number of frequent patterns is 966 at support threshold of 0.5%, 12,625 at 0.25%, 601,936 at 0.195%, and 114,220,668 at 0.15%.

Table 1. Basic features of datasets.

	Trans. Number	Dist. Items	Max. Trans. Size	Aver. Trans. Size
BMS-POS	515,597	1,657	164	6.5
BMS-WebView-1	59,602	497	267	2.5
BMS-WebView-2	77,512	3,340	161	5.0
Connect4	67,557	150	43	43.0
IBM Artificial	100k~15m	20,000	72	28.4

## 4.2 Empirical results

In this subsection, we describe the performance of our algorithm versus Apriori, FP-Growth, and H-Mine on the datasets described in the previous section. The performance measure was the execution time of the algorithms on the datasets with different support threshold. The execution time only includes the disk reading time (scan datasets) and CPU time, but excludes disk writing time (output patterns) in order to reduce the influence of relatively slow speed of disk writing.

Figure 11 through 15 show the performance curves for the four algorithms on the five datasets respectively. The vertical axis is on a logarithmic scale. As we can see, the OpportuneProject algorithm outperforms the other three algorithms on all datasets. The performance improvements of OpportuneProject over other algorithms were significant at reasonably low support thresholds.

For the BMS-POS, at the support thresholds over 0.4%, where the number of frequent patterns is under 6,656, the four algorithms

have the same performance. FP-Growth behaves the same as OpportuneProject when the support threshold is over 0.1%. When the support threshold decreases under 0.1%, the performance gap becomes outstanding. At the reasonable low support threshold of 0.04%, OpportuneProject requires 34 seconds, whereas FP-Growth requires 72 seconds, H-Mine requires 277 seconds, and Apriori requires 781 seconds. At the even lower support threshold of 0.02%, OpportuneProject requires 52, while FP-Growth requires 215 seconds, H-Mine requires 727 seconds, and Apriori requires 2012 seconds. The rankings of algorithms are OpportuneProject > FP-Growth > H-Mine > Apriori.

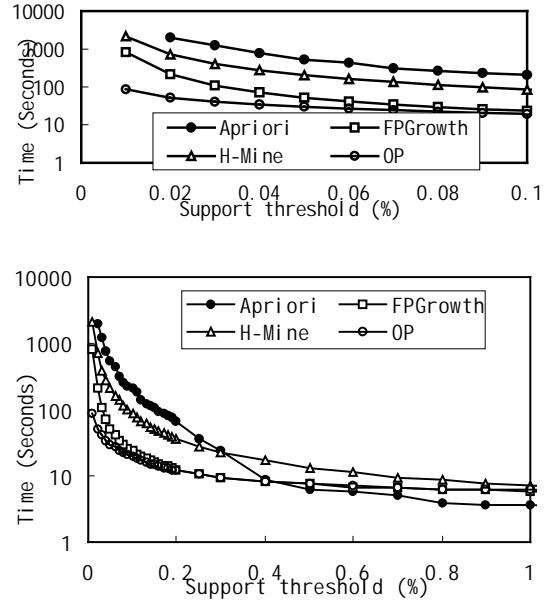


Figure 11. Computational performances on BMS-POS

On BMS-WebView-1 and BMS-WebView-2, the algorithms' rankings are OpportuneProject > H-Mine > Apriori > FP-Growth. When the support threshold is large, such as 0.1% for BMS-WebView-1 and 0.3% for BMS-WebView-2, three algorithms have almost the same execution time, but that of FP-Growth is one order of magnitude greater than others. OpportuneProject is one order of magnitude faster than H-Mine, nearly two order of magnitude faster than Apriori, and far over two orders of magnitude faster than FP-Growth for support threshold under 0.06% on BMS-WebView-1, under 0.05% on BMS-WebView-2.

For connect4, the execution times of H-Mine and Apriori are basically in the same order. OpportuneProject is over three order of magnitude more efficient than H-Mine and Apriori on Connect4. And the execution times of OpportuneProject and FP-Growth are nearly the same order when the support threshold are large. However, the performance gap becomes significant when the support threshold reaches below 80%. For example, OpportuneProject finishes in 5 seconds while FP-Growth runs over 27 seconds for the support level of 70%. And the performance gap between OpportuneProject and FP-Growth is a factor over 20 for support threshold less than 60%.

For IBM Artificial, T25I20D100kN20kL5k, algorithms' ranking are OpportuneProject > H-Mine > FP-Growth > Apriori. When the support threshold is under 0.3%, all algorithm have the same

performance because the maximum pattern length is 1. OpportuneProject outstrips the other three algorithms for support threshold over 0.3%. At the support threshold of 0.195, a reasonably low one where there is 601,936 frequent patterns, OpportuneProject requires 4 seconds, while H-Mine requires 30 seconds, FP-Growth requires 83 seconds, and Apriori requires 450 seconds. When the support threshold decreases to an even lower level, improvements of OpportuneProject are more striking.

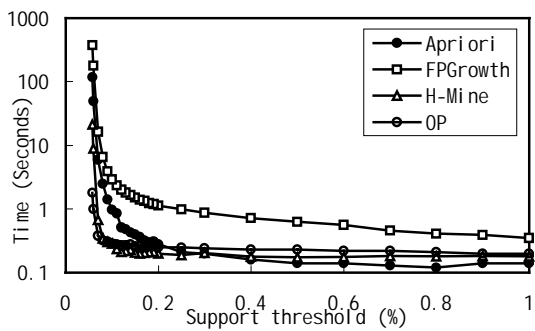
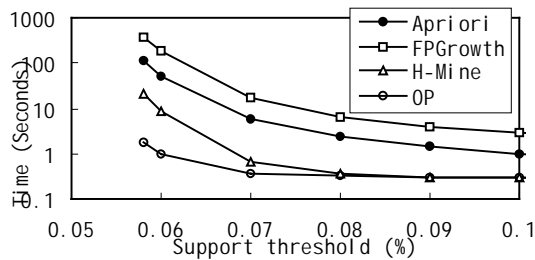


Figure 12. Computational performances on BMS-WebView-1

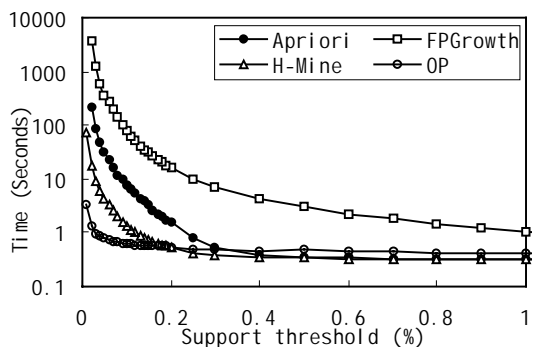
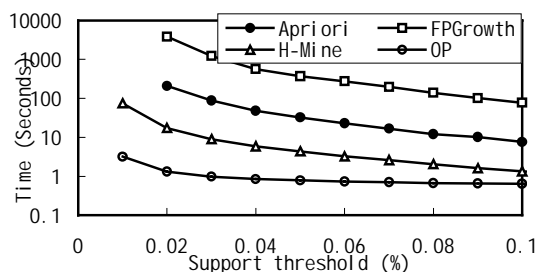


Figure 13. Computational performances on BMS-WebView-2

The performance results we discussed so far are all under the case that there is enough memory for all algorithms.

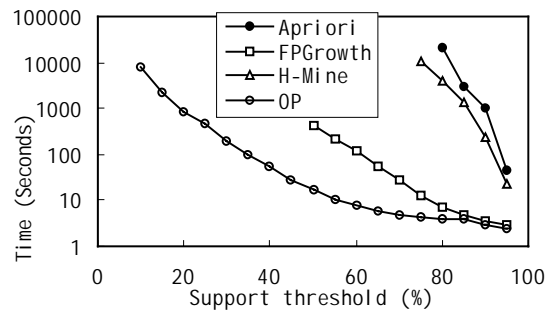


Figure 14. Computational performances on Connect4

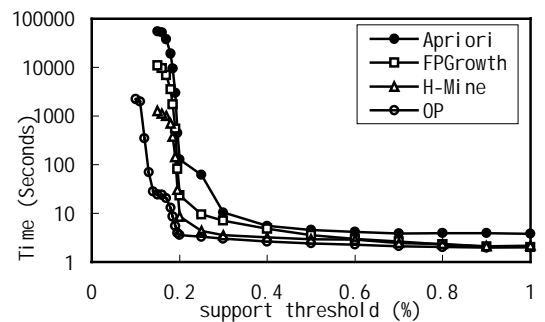
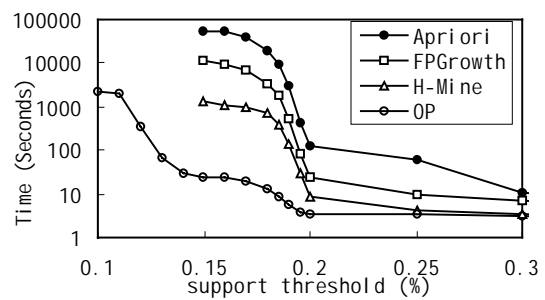


Figure 15. Computational performances on T25I20D100kN20kL5k

### 4.3 Scale up experiments

To test the efficiency and scalability of the algorithms on mining very large databases, we have tested OpportuneProject versus H-Mine, FP-Growth, and Apriori on datasets T25I20N20kL5k, with D100k, D1m, and D10m at support level ranging from 0.1% to 1%, and for D100k through D15m at support level of 0.2%.

The execution time curves of the four algorithms have similar trends on T25I20D1mN20kL5k and T25I20D10mN20kL5 as on T25I20D100kN20kL5k reported in the last subsection, as the three datasets have the same features although there is a shift in the pattern distributions, for example, there are approximately 600,000 frequent patterns at support threshold of 0.195% on D100k, whereas the support threshold should be 0.19% and 0.188% to discover the same number of frequent patterns on D1m and D10m respectively.

OpportuneProject scales to very large database better than others. The performance improvements are much more dramatic. For example, OpportuneProject's performance improvement factor over H-Mine to discover 600,000 frequent patterns increases from 7 to 20 when the database size increases from D100k to D1m.

The factor of OpportuneProject over Apriori increases from 100 to 3000, whereas OpportuneProject over FP-Growth decreases from 16 to 8 in the same case.

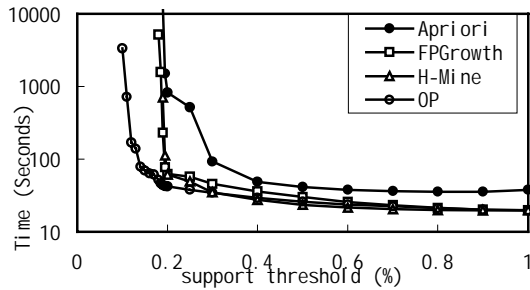


Figure 16 Execution time on T25I20D1mN20kL5k

Figure 18 shows the performance of algorithms on T25I20N20kL5k, while the database size increase from 100k to 15m, at the support threshold of 0.2% where the maximum pattern length is 13, and the number of frequent patterns is around 17K, except 45K for D200k and 99K for D100k. Apriori fails when the database size reaches D2m, and FP-Growth and H-Mine fails at D4m, because they run out of memory. It is very impressive that OpportuneProject scales almost linearly with the database size. For example, OpportuneProject finishes in 551 seconds on D5m, 1295 seconds on D10m, and 1961 seconds on D15m, while the memory consumed is less than 178MB.

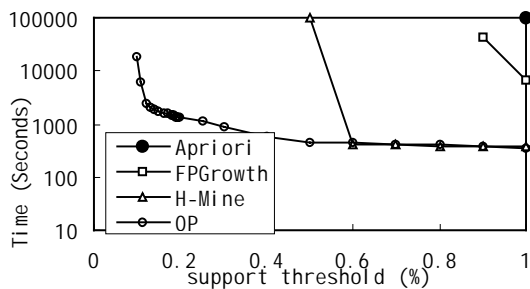


Figure 17 Execution time on T25I20D10mN20kL5k

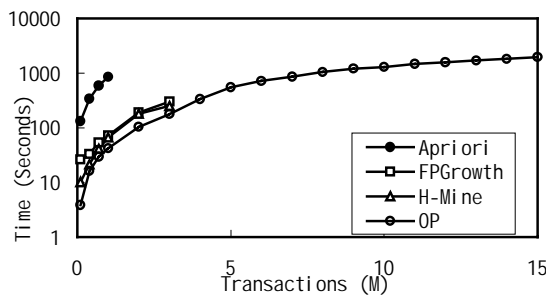


Figure 18 Execution time on T25I20N20kL5k with D100k through D15m at the support threshold of 0.2%

## 5. CONCLUSIONS

In this paper, we propose an efficient algorithm to find complete set of frequent item sets for databases of all features, sparse or dense, and of all sizes, from moderate to very large. This algorithm combines depth first approach with breadth first approach, opportunistically chooses between array-based

representation with tree-based representation for projected transaction subsets, and heuristically employs different projecting methods, such as tree-based pseudo projection, array-based unfiltered projection, and filtered projection, and achieves the maximized efficiency and scalability.

## 6. ACKNOWLEDGEMENTS

We would like to thank Blue Martini Software, Inc. for providing us the BMS datasets.

## 7. REFERENCES

- [1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.
- [2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns, in *Proceedings of SIGKDD Conference*, 2000.
- [3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD'93*, Washington, D.C., May 1993.
- [4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pp. 487-499, Santiago, Chile, Sept. 1994.
- [5] R.J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD'98*, pp. 85-93, Seattle, Washington, June 1998.
- [6] D. Burdick, M. Calimlim, J. Gehrke. MAFLA: A maximal frequent itemset algorithm for transactional databases. In *proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001.
- [7] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, Shalom Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Analysis. In *SIGMOD'97*, 255-264. Tucson, AZ, May 1997.
- [8] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *VLDB'95*, Zurich, Switzerland, Sept. 1995.
- [9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD'2000*, Dallas, TX, May 2000.
- [10] D-I. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *6th Intl. Conf. Extending Database Technology*, March 1998.
- [11] J.S.Park, M.S.Chen, and P.S.Yu. An effective hash based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD*, 175-186, San Jose, CA, Feb. 1995.
- [12] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases, *Proc. 2001 Int. Conf. on Data Mining (ICDM'01)*, San Jose, CA, Nov. 2001.
- [13] Ashok Sarasere, Edward Omiecinsky, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *21st Int'l Conf. on Very Large Databases (VLDB)*, Zurich, Switzerland, Sept. 1995.
- [14] H.Toivonen. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, 134-145, Bombay, India, Sept. 1996.
- [15] Zijian Zheng, Ron Kohavi and Llew Mason. Real World Performance of Association Rule Algorithms. In *Proc. 2001 Int. Conf. on Knowledge Discovery in Databases (KDD'01)*, San Francisco, California, Aug. 2001.
- [16] <http://fuzzy.cs.uni-magdeburg.de/~borgelt/src/apriori.exe>
- [17] <http://www.almaden.ibm.com/cs/quest/syndata.html>
- [18] <http://www.ics.uci.edu/~mllearn/MLRepository.html>