"Performance of Frequent Pattern-Growth hierarchy for bulky and forceful Data Set and advance effectiveness

[1]Mrs. Lubna,[2]Mr.Rahul Moriwal,[3]Dr.Amit khare
Acropolis Institute of Technology and Research, Indore,
Department of computer science engineering.

Abstract: ***FP-growth method is a efficient algorithm to mine frequent patterns, in spite of long or short frequent patterns. By using compact tree structure and partitioning-based, divide-and-conquer searching method, it reduces the search costs substantially. But just as the analysis in Algorithm , in the process of FP-tree construction, it is a strict serial computing process. algorithm performance is related to the database size, the sum of frequent patterns in the database: ω. this is a serious bottleneck. People may think using distributed parallel computation technique or multi-CPU to solve this problem. But these methods apparently increase the costs for exchanging and combining control information, and the algorithm complexity is also greatly increased, cannot solve this problem efficiently. Even if adopting multi-CPU technique, raising the requirement of hardware, the performance improvement is still limited***.

Keyword*:* divide-and-conquer, partitioning-based, parallel, Projection, data mining, AI, Information.

INTRODUCTION: (1). we can create a temp database for storing all the frequent items ordered by the list of frequent items, Lwe call this temp database as Projection Database (or PDB for short), which is used for projecting, reduce the expensive costs of individual node computation.

(2). we can project the PDB, two columns at a time.1 One column (called current column) is used to computer the count of each different item, the other (previous) column is used to distinguish the node's parent node of current column. we can insert one **level** of nodes into the tree at a time, not compute frequent items one by one. Then, the algorithm performance is only related to the **depth** of tree, namely the number of frequent items of the longest transaction in the database η,

(3). because we only project two columns at a time, only save the information of the current nodes and their parent nodes, if there exist the case as follows: the current nodes' parent nodes are identical, but their parent nodes' parent nodes are different, we couldn't judge how to deal with it. If we add their count regarding them as the same node,

***DEFINITION AND BASE FORMULATION** :*

conditional-pattern base (a "sub-database" which consists of the set of frequent items occurring with the suffix pattern), constructs its (*conditional*) FP-tree, and performs mining recursively with such a tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree. Since the frequent item set in any transaction is always encoded in the corresponding path of the frequent-pattern trees, pattern growth ensures the completeness of the result. our method is not *Apriori*-like *restricted generation-and-test* but *restricted test only*. The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most *Apriori*-like algorithms. the search technique employed in mining is a *partitioning-based*, *divide-andconquer method* rather than *Apriori*-like *level-wise generation of the combinations of frequent itemsets*. This dramatically reduces the size of *conditional-pattern base* generated at the subsequent level of search as well as the size of its corresponding *conditional FP-tree*.

A performance study has been conducted to compare the performance of *FP-growth* with two representative frequent-pattern mining methods, *Apriori* (Agrawal and Srikant, 1994) and Tree *Projection* (Agarwal et al., 2001), *FP-growth* outperforms the *Tree Projection* algorithm. our Ftree-based mining method has been implemented in the DBMiner system and tested in large transaction databases in industrial applications. Although *FP-growth* was first proposed briefly in Han et al. (2000), this paper makes additional progress as follows.

– The properties of FP-tree are thoroughly studied. we point out the fact that, although it is often compact, FP-tree may not always be minima.

– Some optimizations are proposed to speed up *FP-growth*, for technique to handle single path FP-tree has been further developed for performance improvements.

– A database projection method has been developed to cope with the situation when an FP-tree cannot be held in main memory—the case that may happen in a very large database.

– Extensive experimental results have been reported. We examine the size of FP-tree as well as the turning point of *FP-growth* on data projection to building FP-tree.

FREQUENT-PATTERN TREE: DESIGN AND CONSTRUCTION

Let $I = \{a1, a2, ...am\}$ be a set of items, and a *transaction database DB*=$T1, T2, . . . , Tn$, where Ti ($i = [1 . . . n]$) is a transaction which contains a set of items in I . The *support*1 (or occurrence frequency) of a *pattern A*, where *A* is a set of items, is the number of transactions containing *A* in *DB*. A pattern *A* is *frequent* if *A*'s support is no less than a predefined *minimum support threshold*, ξ .

A compact data structure can be designed based on the following observations:

(1). Since only the frequent items will play a role in the frequent-pattern mining, it is necessary to perform one scan of transaction database *DB* to identify the set of frequent items (with *frequency count* obtained as a by-product).

(2). If the *set* of frequent items of each transaction can be stored in some compact structure, it may be possible to avoid repeatedly scanning the original transaction database.

(3). If multiple transactions share a set of frequent items, it may be possible to merge the shared sets with the number of occurrences registered as *count*.

database

(1). If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the *count* is registered properly. If the frequent items are sorted in their *frequency descending order*, there are better chances that more prefix strings can be shared. one may construct a frequent-pattern tree as follows. a scan of *DB* derives a *list* of frequent items, (f:4), (c:4), (a:3), (b:3), (m:3), (p:3)(the number after ":" indicates the support), in which items are ordered in frequency descending order. the root of a tree is created and labeled with "*null*".

(1). The scan of the first transaction leads to the construction of the first branch of the tree:(f:1), (c:1), (a:1), (m:1),(p:1). (2). For the second transaction, since its (ordered) frequent item list *f, c, a, b,m* shares a common prefix *f, c, a*with the existing path *f, c, a,m, p* the count of each node along the prefix is incremented by 1, and one new node (b:1) is created and linked as a child of (a:2) and another new node (m:1) is created and linked as the child of (b:1). (3). For the third transaction, since its frequent item list *f, b*hares only the node *f* with the *f* prefix subtree, *f* 's count is incremented by 1, and a new node (b:1) is created and linked as a child of (f:3).(4). The scan of the fourth transaction leads to the construction of the second branch of the tree, (c:1), (b:1), (p:1). (5). For the last transaction, since its frequent item list *f, c, a,m, p*is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

Definition (*FP-tree*). A *frequent-pattern tree* (or *FP- tree* in short) is a tree structure

(1). It consists of one root labeled as "*null*", a set of item-prefix sub trees as the children of the root and a frequent-item-header table.

(2). Each node in the item-prefix sub tree consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node,

node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.

(3). Each entry in the frequent-item-header table consists of two fields,

(1) *item-name*

(2) *head of node-link* (a pointer pointing to the first node in the FP-tree carrying the *item-name*).

Algorithm (FP-tree construction)**.**

Input: A transaction database *DB* and a minimum support threshold ξ.

Output: FP-tree, the frequent-pattern tree of *DB*.

Method: The FP-tree is constructed as follows.

(1). Scan the transaction database *DB* once. Collect *F*, the set of frequent items, and the support of each frequent item. Sort *F* in support-descending order as *FList*, the *list* of frequent items.

(2). Create the root of an FP-tree, *T* , and label it as "null". For each transaction *Trans* in *DB* do the following. Select the frequent items in *Trans* and sort them according to the order of *FList*. Let the sorted frequent-item list in *Trans* be where *p* is the first element and *P* is the remaining list. Call *insert tree*. The function *insert tree* is performed as follows. If *T* has a child *N* such that *N.item-name* = *p.item-name*, then increment *N*'s count by 1; else create a new node *N*, with its count initialized to 1, its parent link linked to *T* , and its node-link linked to the nodes with the same *item-name* via the node-link structure. If *P* is nonempty, call *insert tree*(*P*, *N*) recursively.

COMPLETENESS AND COMPAVTNESS OF FP-TREE

There are several important properties of FP-tree that can be derived from the FP-tree construction process.

Given a transaction database DB and a support threshold ξ . Let F be the frequent items in DB. For each transaction T , freq(T) is the set of frequent items in T , freq(T) = T, F, and is called the frequent item projection of transaction T . According to the Apriori principle, the set of frequent item projections of transactions in the database is sufficient for mining the complete set of frequent patterns, because an infrequent item plays no role in frequent patterns.

Based on the FP-tree construction process, for each transaction in the *DB*, its frequent item projection is mapped to one path in the FP-tree. For a path *a*1*a*2 . . . *ak* from the root to a node in the FP-tree, let *cak* be the count at the node labeled *ak* and *c k* be the sum of counts of children nodes of *ak* . According to the construction of the FP-tree, the path registers frequent item projections of *cak* - *c k* transactions. Therefore, the FP-tree registers the complete set of frequent item projections without duplication. Based on this lemma, after an FP-tree for *DB* is constructed, it contains the complete information for mining frequent patterns from the transaction database. only the FP-tree is needed in the remaining mining process, regardless of the number and length of the frequent patterns. Based on the FP-tree construction process, for each transaction in the *DB*, its frequent item projection is mapped to one path in the FP-tree. For a path *a*1 *a*2 . . . *ak* from the root to a node in the FP-tree, let *cak* be the count at the node labeled *ak* and *c k* be the sum of counts of children nodes of *ak* . the path registers frequent item projections of *cak* - *c k* transactions Therefore, the FP-tree registers the complete set of frequent item projections without duplication. Based on this lemma, after an FP-tree for *DB* is constructed, it contains the complete information for mining frequent patterns from the transaction database. only the FP-tree is needed in the remaining mining process, FP-tree is a highly compact structure which stores the information for frequent-pattern mining. Since a single path "a1→a2 →a3→ a4…→" an in the *a*1-prefix sub tree registers all the transactions whose maximal frequent set is in the form of "a1→a2 →a3→ a4…→ak" for any 1≤k≤n the size of the FP-tree is substantially.

Connect-4 used in *Max Miner* (Bayardo, 1998), which contains 67,557 transactions with 43 items in each transaction, when the support threshold is 50% (which is used in the *Max Miner* experiment.

MINING FREQUENT PATTERNS USING FP-TREE

Construction of a compact FP-tree ensures that subsequent mining can be performed with a rather compact data structure. this does not automatically guarantee that it will be highly efficient since one may still encounter the combinatorial problem of candidate generation if one simply uses this FP-tree to generate and check all the candidate patterns. we study how to explore the compact information stored in an FP-tree, develop the principles of frequent-pattern growth by examination of our running example,

PRINCIPLES OF FREQUENT-PATTERN GROWTH FOR FP-TREE MINING

Property (1) (*Node-link property*)**.** *For any frequent item ai , all the possible pattern containing only frequent items and ai can be obtained by following ai 's node-links, starting from ai 's head in the FP-tree header.*

Property (2) (Prefix path property)**.** To calculate the frequent patterns with suffix ai , only the prefix sub pathes of nodes labeled ai in the FP-tree need to be accumulated, and the frequency count of every node in the prefix path should carry the same count as that in the corresponding node ai in the path.

Property (3) (*Fragment growth*)**.** *Let α be an item set in DB, B be α's conditional pattern base ,β be an item set in B. Then the support of α Uβ in DB is equivalent to the support of β in B.*

TRADITIONAL FREQUENT PATTERN GROWTH ALGORITHM

Let I ={a1, a2, … am} be a **set of items**, and a **transaction database** *DB* =(T1, T2, …Tn), where Ti (i∈[1..n]) is a transaction which contains a set of items in I. Every transaction has a key label, called TID. The **support**1 (or occurrence frequency) of a **pattern** A, which is a set of items, is the number of transactions containing A in *DB*. A is a **frequent pattern** if the support of A is no less than a predefined minimum support threshold ξ. Given a transaction database *DB* and a minimum support threshold, ξ, the problem of finding the complete set of frequent patterns is called the **frequent pattern mining problem**.

(1). Since only the frequent items will play a role in the frequent pattern mining, it is necessary to perform one scan of *DB* to identify the set of frequent items (with frequency count obtained as a by-product).

(2). If we store the set of frequent items of each transaction in some compact structure, it may avoid repeatedly scanning of *DB*.

(3). If multiple transactions share an identical frequent item set, they can be merged into one with the number of occurrences registered as count. It is easy to check whether two sets are identical if the frequent items in all of the transactions are sorted according to a fixed order.

(4). If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the count is registered properly. If the frequent items are sorted in their frequency descending order, there are better chances that more prefix strings can be shared. Based on the above observations, we can get the **definition of FP-tree**:

(1). It consists of one **root** labeled as "null", a set of **item prefix subtrees** as the children of the root, and a **frequent-item header table**.

(2). Each node in the **item prefix subtree** consists of three fields: item-name, count, and node-link, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.

(3). Each entry in the **frequent-item header table** consists of two fields, (1) item-name and (2)head of node-link, which points to the first node in the FP-tree carrying the item-name. Based on this definition, we have the following FP-tree construction algorithm.

Algorithm (FP-tree construction)

Input: A transaction database *DB* and a minimum support threshold ξ.

Output: Its frequent pattern tree, FP-Tree

Method: The FP-tree is constructed in the following steps.

(1). Scan the transaction database *DB* once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.

(2). Create the root of an FP-tree, T, and label it as "null", for each transaction in *DB* Select and sort the frequent items in transaction according to the order of L. Let the sorted frequent item list in transaction be where p is the first element and P is the remaining list. Call **insert tree .**

Function insert tree

If T has a child N such that N.item-name = p.item-name

Then increment N's count by 1;

Else do {create a new node N;

N's count = 1;

N's parent link be linked to T;

N's node-link be linked to the nodes with the same item-name via the node-link

structure;}

If P is nonempty, **Call insert tree (P, N)**.

CONSTRUCTING FP-TREE GROWTH USING PROJECTION

FP-growth method is a efficient algorithm to mine frequent patterns, in spite of long or short frequent patterns. By using compact tree structure and partitioning-based, divide-and-conquer searching method, it reduces the search costs substantially. But just as the analysis in Algorithm and in the process of FP-tree construction, it is a strict serial computing process. algorithm performance is related to the database size, the sum of frequent patterns in the database: ω. People may think using distributed parallel computation technique or multi-CPU to solve this problem. But these methods apparently increase the costs for exchanging and combining control information , cannot solve this problem efficiently. Even if adopting multi-CPU technique, raising the requirement of hardware, (1) we can create a temp database for storing all the frequent items ordered by the list of frequent items L. we call this temp database as Projection Database (or PDB for short), which is used for projecting, reduce the expensive costs of individual node computation.

(2) we can project the PDB, two columns at a time. One column is used to computer the count of each different item, the other (previous) column is used to distinguish the node's parent node of current column. By this way, we can insert one **level** of nodes into the tree at a time, not compute frequent items one by one. Then, the algorithm performance is only related to the **depth** of tree, namely the number of frequent items of the longest transaction in the database η, not the sum of frequent items in the database.

(3) because we only project two columns at a time, only save the information of the current nodes and their parent nodes, if there exist the case as follows: the current nodes' parent nodes are identical, but their parent nodes'

PROPOSED ALGORITHM:

Algorithm PFP-tree construction

Input: A transaction database DB and a minimum support threshold ξ.

Output: PFP-tree

Method:(1). Scan the transaction database *DB* once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, (2). Select and sort the frequent items in transaction according to the order of L, the result is saved in the PDB.

(3). Create the root of an FP-tree, T, and label it as "null". Let column number in PDB be j, the initial value of j is 1.

If j = 1

The process is implemented as follows: first project the column (j-1) and column (j), then add 1 to j, and project column (j-1) and column (j) circularly, and so on, until project the last column of PDB.

Then do { Project the column (1), collect the set of frequent items and their supports, let the result be [q:n], where q is the frequent item, n is the count;

Insert these nodes as the root's **child** nodes into the PFP-tree. **}**

Else do { (1) Project both parent column (j-1) and current column (j), compare the set of

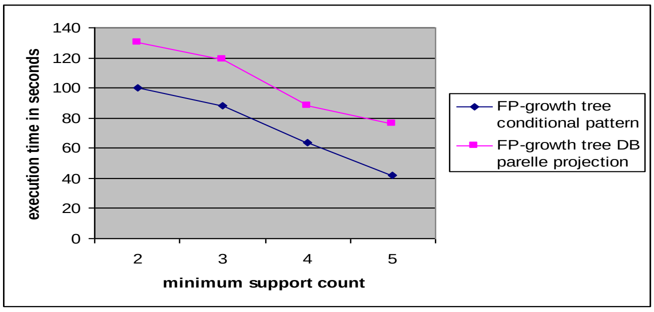
binary-frequent items and collect their supports1. Let the result be [px, q:n], where p is the parent frequent item of column(j-1), x is p's TAI if it has (if p has no TAI, then let x be null) and q is the current frequent item of column (j), n is the count;

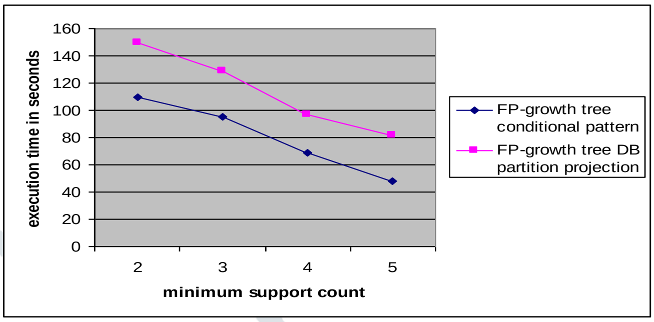
(2) Compare the result sets of [px, q:n], if their current frequent item name, q are identical, then add each px as its TAI to q, let the result be [px, qy:n], where y=px; **Else do nothing**.

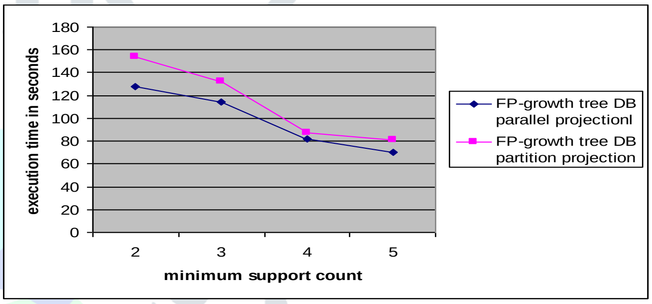
(3) Insert the nodes [qy:n] or [q:n] as the child nodes of px into the PFP-tree and let their node-link be linked to the nodes with the same item-name via the node-link structure.

(4). Delete all the TAI in the PFP-tree and PDB.

COMPARIOSION ON THE BASIS OF VARING MINIMUM SUPPORT EXECUTION TIME

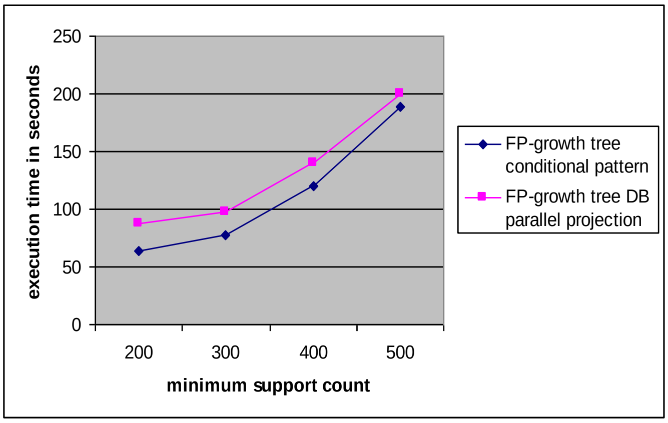




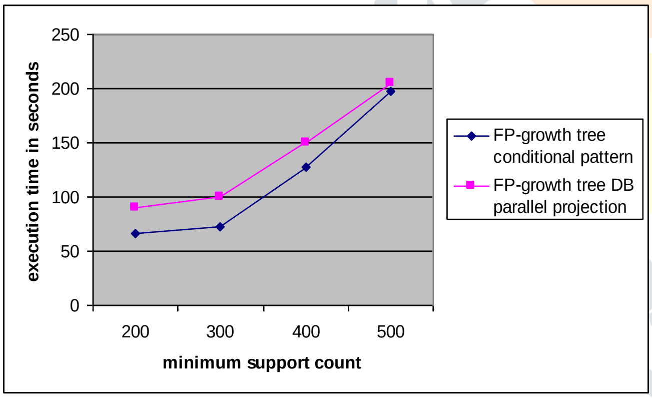


COMPARIOSION ON THE BASIS OF VARING NUMBER OF RECORDSAND EXECUTION TIME

Number of records	Time taken to execute (In millisecond) **FP-GROWTH Tree with conditional**	Time taken to execute (In millisecond) **FP-GROWTH Tree with Data base Parallel projection algorithm**
200	64	87
300	78	97
400	120	140
500	189	200



Number of records	Time taken to execute (In millisecond) **FP-GROWTH Tree with conditional**	Time taken to execute (In millisecond) **FPGROWTH Tree with Data base Partition projection algorithm**
200	66	90
300	72	100
400	127	150
500	197	205



Conclusion: Since a transaction is projected to only one projected database at the database scan, after the scan, the database is partitioned by projection into a set of projected Databases, and hence it is called *partition projection*. The projected databases are mined in the reversed order of the *list of frequent items*. the projected database of the least frequent item is mined first, and so on. Each time when a projected database is being processed, to ensure the remaining projected databases obtain the complete information, each transaction in it is projected to the aj - projected database, where aj is the item in the transaction such that there is no any other item after aj in the list of frequent items appearing in the transaction. The partition projection process for the database. The advantage of partition projection is that the total size of the projected databases at each level is smaller than the original database, and it usually takes less memory and I/Os to complete the partition projection. the processing order of the projected databases becomes important, and one has to process these projected databases in a sequential manner. during the processing of each projected database, one needs to project the processed transactions to their corresponding projected databases, which may take some I/O as well. Nevertheless, due to its low memory requirement, partition projection is still a promising method in frequent pattern mining

References:

1. Etzioni, O. (1996). The world-wide Web: quagmire or gold mine? Communications of the ACM, 39 (11), 65–68.
2. Kosala, R and Blockeel, H. (2000). Web mining research: a survey. SIGKDD Explorations, July, 2 (1), 1-15.
3. Brin, S and Page,L. (1998). The anatomy of a large-scale hyper-textual Web search engine. Computer Networks and ISDN Systems, 30 (1–7), 107–117.
4. J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential Pattern Mining using A Bitmap Representation, SIGKDD'02, 2002.
5. J.R. Punin, M.S. Krishnamoorthy, and M.J. Zaki. (2001). LOGML - Log Markup Language for Web Usage Mining, in WEBKDD Workshop 2001: Mining Log Data Across All Customer TouchPoints (with SIGKDD01), San Francisco, August, pp. 88–112.
6. Srivastava J , Cooley R , and Mukund Deshpanda. (2000). Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. SIGKDD Explorations, 1 (2), 12-23.
7. Zhang Huiying and Liang, Wei. (2004). An intelligent algorithm of data pre-processing in Web usage mining, Proceedings of the World Congress on Intelligent Control and Automation (WCICA), v 4, WCICA, p3119-3123.
8. Long Wang. (2004). Christoph Meinel. Behaviour Recovery and Complicated Pattern Definition in Web Usage Mining. Web Engineering: 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, pp. 531 – 543.
9. Guo, Jiayun, Keelj, Vlado, and Gao, Qigang. (2005). Integrating Web Content Clustering into Web Log Association Rule Mining, Advances in Artificial Intelligence: 18th Conference of the Canadian Society for Computational Studies of Intelligence, Canada, May 9-11, pp. 182
10. Agrawal, R and Srikant, R. (1994). Fast algorithms for mining association rules, Proc. of the 20th international Conference on very large database, Chile, 487-499.
11. Park J S, Chen M -S, and Yu P S. (1995). An effective Hash-based algorithm for mining association rules, Proceedings of 1995 ACM-SIGMOD International Conference on Management of Data (SIGMOD'95). San Jo se, CA, 175-186.
12. Savasere A ,Omiecinski E, and Navathe S. (1995). An efficient algorithm for mining association rules in large databases . VLDB'95 , 432-443.
13. Toivonen H. (1996). Sampling Large Databases for Association Rules, Proceedings of 22th VLDB Conf. Bombay, India, 134-145.

14. R. Agrawal, and R. Srikant, "Mining Sequential Patterns, " ICDE, 1995.
15. J. Han, J. Pei, B. Mortazavi-Asi, Q. Chen, U. Dayal, M. C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining, " SIGKDD'00, 2000
16. J. Pei, J. Han, B. Mortazavi-Asi, H. Pino, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," ICDE'01, 2001.
17. H. Pinto, J. Han, J. Pei, K. Wang, "Multi-dimensional Sequence Pattern Mining, " CIKM'01, 2001.
18. R. Srikant, and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements, EDBT, " 1996.
19. M. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential pattern mining with regular expression constraints, " VLDB'99, 1999.
20. J. Pei, J. Han, J. Wang, H. Pinto, Q. Chen, U. Dayal, M. C. Hsu, "Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, " IEEE Transactions on Knowledge and Data Engineering, Oct, 2004.
21. M. Seno and G. Karypis, "SLPMiner: An Algorithm for Finding Frequent Sequential Patterns Using Length-Decreasing Support Constraints," ICDM'02, 2002.
22. J. Pei, J. Han, and W. Wang, "Mining Sequential Patterns with Constraints in Large Databases," ACM CIKf, Nov. 2002.
23. J. Wang, and J. Han, "BIDE: Efficient Mining of Frequent Closed Sequences, ICDE'04, 2004.
24. X. Yan, J. Han, R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets, " SDM'03, 2003.
25.]M. Zaki, "SPADE: An efficient algorithm for mining frequent sequences. Machine Learning, " 2001.
26 U. Yun and J.J. Leggett, "WSpan: Weighted Sequential Pattern Mining in Large Sequence Databases," *Proc. Of the Third Int'l Conf. on IEEE Intelligent Systems*, Sep. 2006, pp. 512-517.
27. U. Yun and J.J. Leggett, "WFIM: Weighted Frequent Itemset Mining with a Weight Range and a Minimum Weight," *Proc. Of the Fifth SIAM Int'l Conf. on Data Mining*, Apr. 2005, pp. 636- 640.
28. U. Yun and J.J. Leggett, "WLPMiner: Weighted Frequent Pattern Mining with Length-Decreasing Support Constraints," *Proc. Of* the *9th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD`05)*, May 2005, pp. 555-567.
29. U. Yun, "Mining Lossless Closed Frequent Patterns with Weight Constraints," *Knowledge Based Systems*, vol. 20, Feb. 2007, pp. 86-97.
30. Mahdi Esmaeili and Fazekas Gabor, "Finding Sequential Patterns from Large Sequence data," IJCSI, vol. 7, Issue1,Jan2010, pp. 43-46.