An Efficient Algorithm for Distributed Incremental Updating of Frequent Item-Sets on Massive Database

Jiangtao Qiu, Changjie Tang, Lei Duan, Chuan Li, Shaojie Qiao, Peng Chen, and Qihong Liu

School of Computer Science, Sichuan University, Chengdu, China
{qiujiangtao, tangchangjie}@cs.scu.edu.cn

Abstract. Incremental updating of frequent item-sets on a database includes three problems. In this paper, these problems are explored when database stores massive data. The main contributions include: (a) introduces the concept of Interesting Support Threshold; (b) proposes Frequent Item-sets Tree (FITr) with compact structure; (c) proposes and implements algorithm FIIU for frequent item-sets incremental updating; (d) in order to further improve performance, proposes the algorithm DFIIU for distributed incremental updating of frequent Item-sets on massive database; (e) gives extensive experiments to show that FIIU and DFIIU algorithms have better performance than traditional algorithm on massive database when the number of items is less.

1 Introduction

Association Rules Mining (ARM) is an active research area on Data Mining. Since its introduction in [1], there have been extensive studies on efficient frequent item-set mining methods. The process of mining association rules consists of two main steps: (1) Find the frequent item-sets under support threshold; (2) Generate strong association rules from the frequent item-sets. Most studies focus on step 1 because step 1 is more difficult than step 2. However, finding frequent item-sets by simply re-executing mining on the whole database will be very low efficient when new data were inserted into database or support threshold changed, especially for database stored massive data. Although parallel algorithm and sampling algorithm have been proposed to find association rules on massive data, it is obvious that they are not suitable for the above problem. Therefore, using the prior knowledge to find out new item-sets on the updated database, so called *Incremental Updating of Frequent Item-sets*, has become a hot topic. It involves three problems:(1) Finds out frequent item-sets under new support threshold without database updating; (2) Finds out frequent item-sets when database was updated, but support threshold was unchanged; (3) Finds out frequent item-sets when database was updated and support threshold was changed.

There have been many incremental mining algorithms being proposed, and they have been proved to be good at medium scale database. However, these algorithms showed one or several neglects: (1) only solved one of three problems; (2) Needed to scan database repeatedly; (3) Although some algorithm scanned database once, they didn't prove good performance on the massive database.

L. Feng et al. (Eds.): WISE 2006 Workshops, LNCS 4256, pp. 61 – 72, 2006.

In order to solve the three problems, our study makes following contributions based on idea of not scanning original database.

1) Introduces the concept of *Interesting Support Threshold* Sup_{min}.
2) Proposes a prefix tree with compact structure, called Frequent Item-sets Tree.
3) Proposes an efficient algorithm FIIU (Frequent Item-sets Incremental Updating).
4) In order to further improve performance, the algorithm DFIIU (Distributed Frequent Item-sets Incremental Updating) is proposed for incremental mining frequent Item-sets on massive databases with multiple computers.

The remaining of this paper is organized as follows. Section 2 gives a briefly introduction to related works. Section 3 revisits description of the problem. Section 4 proposes algorithm FIIU. Section 5 proposes algorithm DFIIU. Section 6 gives a thorough performance study in comparison with FP-Growth algorithm. Section 7 summarizes our study.

2 Related Works

Some efficient algorithms have been proposed for finding frequent item-sets. Apriori[2] employs iterative approach known as level-wise search, where k-item-sets are used to explore k+1-item-sets. Apriori based DHP[3] uses hash table to improve performance on mining frequent item-sets. Partition[4] employs partition strategy to find frequent item-sets with only twice database scans. Han's FP-Growth[5] may avoid candidates generation and utilizes compact data structure, called *frequent pattern tree*, to generate frequent item-sets with divide-and-conquer strategy.

To address problem of *incremental updating of frequent item-sets*, Cheung and Li first propose FUP[4] and FUP2[5]. However, these algorithms only focus on second and third problems of *incremental updating of frequent item-sets*, and need to scan the entire database several times. In [6], a CAST tree is proposed. CAST may be employed to find frequent item-sets when support threshold changes. FIUA2[7] and IM[8] are FP-tree based algorithm. They incrementally mine frequent item-sets with strategy that use new data to extend old FP-tree. However, the two algorithms do not analysis performance on massive data.

Main approaches of mining frequent item-sets on massive data include parallel-based approach and the sampling-based approach. CD[9] is an Apriori-like parallel algorithm. DD[9] and IDD[10] divide candidate set into several sub-sets, and then send each sub-set to one processor. Quick-Mining[11] is a sampling-based algorithm.

3 Problem Descriptions

Let I_k be an item, and $I=\{I_1,I_2,\ldots, I_m\}$ be a complete collection of items. A transaction database $DB=\{T_1, T_2,,\ldots, T_n\}$ is a collection of transactions, where T_i (i=1,…,n) is a transaction which contains items in *I*. *Sup* denotes support threshold. A item-set is called frequent item-set if its support count is not less than $Sup\times|DB|$ where $|DB|$ is the number of transactions in database *DB*.

Given a transaction database *DB* and a threshold *Sup*, the problem of *frequent item-set mining* is to mine complete collection of frequent item-sets on *DB* with support threshold *Sup*.

Let *db* be the collection of new transactions, and |*db*| be the numbers of new transactions, the updated database $U=DB \cup db$. The problems of *incremental updating of frequent item-sets* include: (1) Find out frequent item-sets $IS_{sup'}(DB)$ if support threshold *Sup* is changed to *Sup*', but database is not updated, *U=DB*. (2) Find out frequent item-sets $IS_{sup}(U)$ if database is updated, U= DB ∪ db, but support threshold is not changed. (3) Find out frequent item-sets $IS_{sup'}(U)$ when database was updated, $U= DB \cup db$, and support threshold is changed to *Sup*'.

The problem of *Distributed Incremental Updating of Frequent Item-sets* is to find out all frequent item-sets by utilizing distributed system when support threshold is changed and database is updated.

4 Incremental Updating of Frequent Item-Sets

In order to not scan original database DB when incrementally mining frequent item-sets, in our study, all item-sets in *DB* will be stored. However the number of item-sets in DB may be large, especially when encountering long transactions in very large DB. Therefore, firstly, we introduce a concept, called Interesting Support Threshold Sup_{min} in section 4.1. For each item-set *A* in DB, if *A.count* is not less than $Sup_{min} \times |U|$, *A* is a frequent item-set. Furthermore, all item-sets in *DB* may be divided into two categories: frequent item-sets and non-frequent item-sets. All frequent item-sets are stored in a prefix tree with compact structure, called FITr, which is defined in section 4.2. Then all reduced non-frequent item-sets are stored in two-level hash-indexed linked-lists, called THL, which is defined in section 4.3. Finally, FITr and THL are materialized on disk.

For first problem of *Incremental Updating of Frequent Item-sets*, FITr is firstly read from disk when support threshold is changed to $Sup (Sup > Sup_{min})$. Then FITr is traveled to find out each item-set whose support count is not less than $Sup \times |U|$.

For the second and the third problems, firstly, all item-sets in *db* will be found out. Then store them into a data structure, called WITs, which is defined in section 4.3. Finally, utilizes Algorithm FIIU, described in 4.4, to realize incremental updating of frequent item-sets.

4.1 Interesting Support Threshold

It is not necessary to find out item-sets under all support thresholds when apply mining association rules to industry. The smaller support threshold is, the larger the number of frequent item-sets is, and then the more difficult generating strong association rules is. Therefore, it can be concluded that generating significant strong association rules is impossible when support threshold is too small. Many criteria [12] are proposed to mine significant association rules. In this study, we defined *Interesting Support Threshold* according to these criteria.

Definition 1(Interesting Support Threshold Sup_{min}). Let *DB* be database and *R* be a set of criteria for mining association rules. Sup_{min} is called *interesting support threshold*

if Sup_{min} is minimal support threshold that satisfied R and may generate significant strong association rules on DB.

4.2 Frequent Item-Sets Tree

To design a compact data structure to store frequent item-sets, let's examine an example firstly.

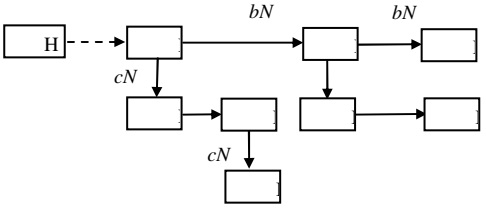

Fig. 1. Example of frequent item-set tree

Example 1. Let $\{I_1,I_2\},\{I_1,I_3,I_4\},\{I_2,I_3\},\{I_2,I_4\},\{I_3\}$ be frequent item-sets. They are stored in frequent item-sets tree (or FITr in short). FITr is illustrated in Fig.1. Node I_j indicates that item I_j is stored in the node.

A *frequent item-sets tree* may be designed as follows based on *example 1*.

Definition 2(Frequent Item-sets Tree FITr). A *Frequent Item-sets Tree* (or *FITr* in short) is a tree structure defined below.

1) FITr consists of one root labeled as *head*. Each node in the FITr consists of four fields: *item*, *count*, *cNode, bNode*, where *item* registers item the node saved, *count* registers support count of an item-set, *cNode* is a pointer which point to first node in *child level*, *bNode* is also a pointer which point to brother node.
2) Let $node_1$ and $node_2$ be nodes in FITr. We call that $node_1$ and $node_2$ are in same level if $node_1.bNode=node_2$ or $node_1.bNode\ldots..bNode=node_2$.
3) Let $node_1$ be a child node of root, we call that $node_1$ is a node in first level in FITr. Starts off from $node_1$, then goes to one of its children node, and go on by this way if $node_n$ may be reached at last, we call that there exists a path between $node_1$ and $node_n$. The set of items of all nodes in the path represents a frequent item-set, and *count* field of $node_n$ represents support count of the frequent item-set.

Example 2. In our running example shown in Fig.1, I1 and its brother node I2, I3 are in same level. Path of node I1 to node I4 is I1 $\rightarrow$ I3 $\rightarrow$ I4. All nodes in the path represent an item-set A= $\{I_1, I_3, I_4\}$. Value of *count* field of node I4 denotes support count of A.

In our study, each item is given a unique integer number *id*.

Property 1. Let *id1* and *id2* be integer number of *node.item* and *node.bnode.item* respectively. Then *id1* is less than *id2*.

Property 2. Let i*d1* and *id3* be integer number of *node.item* and *node.cnode.item* respectively. Then *id1* is less than *id3*.

Property 1 ensures that nodes in same level will be sorted in ascending order by their integer number. Furthermore, cost of inserting operation and searching operation in

FITr will be reduced efficiently. *Property 2* ensures that those frequent item-sets, which have same prefix, share same prefix sub-path in FITr. Furthermore, FITr may store frequent item-sets compactly.

Algorithm 1 InsertFIT: inserting frequent item-sets to FITr

Input: item-set *A*, FITr

Symbols: *A.count* denotes support count of the item-set; *item(i)* denotes No.i item in item-set *A*; *t* denotes the number of items in item-set A.

Method:

(1) Sort(A);
(2) for(int i=1, *node*=*head*; i<=t; i++){
(3) wNode=FindNode (*node*,item(i));
(4) if (wNode=null) wNode=Insert(item(i));
(5) *node*=wNode;}
(6) *node*.count=*node*.count+A.count;

For all item-sets in FITr, their items should be ranked in ascending order by *id* as shown in *Property 2*. Therefore, in step 1, items of the item-set are ranked before the item-set is inserted into FITr. The function FindNode(node,item(i)) search the node whose item field is same with item(i) in *node*'s child level. If the node is found, return the node, otherwise return *null.* If *null* is returned, in step 4, a new node, whose *item* field is set to item(1) and *count* is set to zero, will be generated, then inserted into *node*'s child level by *Property 1*. After the last node of item-set being inserted into FITr, in step 6, *count* of the node will be modify because the *count* of the last node should be support count of *A*.

According to algorithm 1, support count of item-set *A* will be updated correctly when insert *A* to FITr and there exists *A* in FITr.

Theorem 1. (Compactness of FITr) Let *I* be collection of items, $I=\{I_1,I_2,\ldots,I_m\}$, *ID(x)* be unique integer number given$\in$ to $x(x\in I)$, *A1* and *A2* be item-set, and *Path(A)* be path of item-set *A* in FITr. We call that $Path(A1)\subset Path(A2)$ if $A1\subset A2$ and $(\forall x)(\forall y)(x\in A2\wedge x\notin A1\wedge y\in A1\wedge ID(x)>ID(y))$.

Proof. We assume that *A1* has been inserted into FITr and the path of *A1* is from *node1* located in first level to *node2*. Let $A2=A1\cup\{I_m\}$ and $ID(I_m)>ID(x)$ $(x\in A1)$. According to *Property 2*, the order of all items of *A2* being inserted into FITr is items in $A1\cap A2$ at first, then item I_m. According to step 3 and step 5 in Algorithm 1, the path of *A1* will be found after all items in $A1\cap A2$ are inserted into FITr. A new node *node3* (*node3.item*=I_m) will be inserted at child level of *node2*. Name the path of node3 to node1 as the path of *A2*, denoted as *Path(A2)*. It can be concluded that $Path(A1)\subset Path(A2)$.

4.3 Storing of Non-frequent Item-Sets and Updating of FITr

In our study, a two-level hash indexed linked-lists structure, called THL, is adopted to store non-frequent item-sets. Each linked-list is given a unique name, called *class-name*. Each node in linked-list consists of three fields: *count*, *next*, *itNum*, where

count registers support count of a item-set stored in linked-list, *next* points to next node, and *itNum* store the integer value gotten by reducing the item-set.

Process of Item-set reduction includes follow steps. (1) Derive each item of an item-set and its integer number *id*. (2) Let an integer, called *num,* be zero, then set No. *id* bit of *num* to 1. By the reduction, different item-set may be represented by different integers. For example, let {I1, I4, I6} be an item-set, 1, 4, 6 be integer number of I1, I4 and I6 respectively. As a result, *itNum*=41.

By summing up *id* of all items in item-set, we can get an integer, denoted as *a*. Then, denote *a* modulo 10 as *key1*. Let *b* be the base *e* logarithm of *a*. Denote value of second place after decimal point of *b*, as *key2*.

Structure of THL is shown in Fig.2. T1 and T2 are *class-name* of linked-list. First level of two-level hash index uses *key1* as hash key; another uses *key2* as hash key. When an item-set is given, the *class-name* of linked-list the item-set belongs to can be gotten by two-level hash index.

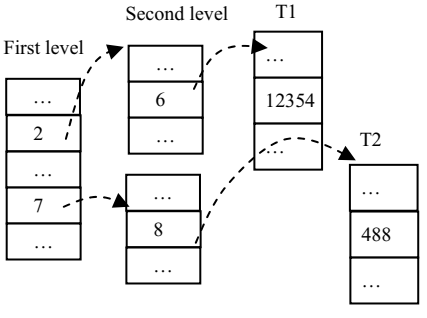

Fig. 2. Two-level hash indexed linked-lists

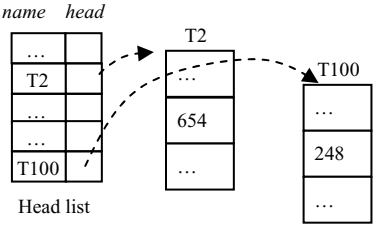

Fig. 3. Data structure of WITs

To insert the item-set to a linked-list, firstly, reduce item-set to an integer. Then generate a node of linked-list. Finally, insert the node to linked-list by *itNum* ascending order. The *count* of the node would be added if there exists node that has same *itNum* with the new node.

According to the above data structure, all non-frequent item-sets may be saved to one hundred of linked-lists. Each linked-list will be materialized in disk by its *class-name*.

In our study, a data structure, called WITs, is adopted to store temporary data during incremental updating of frequent item-sets. WITs includes a head list and linked-lists, whose structure is same with linked-lists in THL. The node in head list consists of two fields: *name* and *head*, where *name* registers *class-name* of a linked-list, *head* is pointer that points to a linked-list. Fig.3 shows structure of WITs.

Before inserting a item-set to WITs, get *class-name* of the linked-list that the item-set belongs to by using the two-level hash index, then gets the linked-list by searching head list of WITs, and inserts item-set to the linked-list.

Algorithm 2 GetFITr: Updating of FITr
Input: FITr, WITs, THL, Sup_{min}
Output: Incremental updated FITr

Method:
(1) While(traveling FITr){
(2) get item-set *A* from FITr;
(3) InsertSI(*A*, WITs);}
(4) FITr=$\varnothing$;
(5) while(there exist unprocessed linked-lists in WITs and THL){
(6) wl=GetWL(WITs);
(7) nl=GetNL(THL, wl);
(8) nl=Merge(wl, nl, FITr);
(9) WriteObject(nl);}
(10) return FITr;

From step 1 to 3, algorithm *GetFITr* travels FITr. Then inserts all item-sets stored in FITr to WITs. Function *InsertSI(A, WITs)* inserts item-set *A* to a linked-list in WITs. Function GetWL (WITs) gets an unprocessed linked-list in WITs, denoted as *wl*. Function GetNL (THL, wl) gets a linked-list of THL stored in disk, denoted as *nl*, which has same *class-name* with *wl*. Function Merge(*wl*, *nl*, FITr) merges *wl* to *nl*. Nodes having same itNum in both *nl* and *wl* sum up *count* of nodes. Then those nodes, whose counts are not less than Sup_{min}, are inserted into a new FITr and deleted from *nl*. In step 9, stores *nl* to disk and free *nl* and *wl* from main memory.

Lemma 1. Assume that a new FITr and a new THL can be derived after running algorithm *GetFITr*. There exists no same item-set in both the FITr and the THL, and the count of item-set is correct.

Proof. i: The two-level hash index ensures that an item-set will be given a unique *class-name*. Therefore, it is impossible that an item-set may emerge on two linked-lists with different *class-name* in either THL or WITs. ii: Operation of inserting the item-set to linked-list ensures count of node in linked-list to be added when inserts same item-set to a linked-list. iii: According i and ii, the count of item-set may be added correctly when the item-set emerge in both THL and WITs because all same item-sets will be inserted into same linked-list in merging operation of *Algorithm 2*. iv: There will not be item-sets emerging in both FITr and THL after running *algorithm 2* because an item-set will be inserted into FITr or THL by being or not being frequent.

Lemma 1 means that algorithm *GetFITr* can create a new FITr that is incrementally updated correctly.

4.4 Algorithm of Frequent Item-Set Incremental Updating

To find frequent item-sets under new support threshold *Sup* ($Sup > Sup_{min}$), first step is to read FITr from disk. Then travels FITr to find item-sets whose support count is not less than $Sup \times |U|$.

Algorithm 3 is used to find frequent item-sets when database was updated and support threshold was changed.

Algorithm 3 FIIU: Incremental updating of Frequent Item-sets
Input: *db*
Output: FITr, THL

Method:
(1) WITs=General_FI(*db*);
(2) FITr=ReadFIT(); // FITr is gotten from disk.
(3) FITr=GetFITr(FITr, WITs, THL);

In first step of algorithm 3, all item-sets of *db* are found, and then are inserted to WITs. *General_FI* is a general algorithm that mines frequent item-sets in dataset, such as Apriori[2] and FP-growth[5]. Step 3 uses algorithm 2 to generate the new FITr, which store all frequent item-sets of updated database.

Lemma 2. Given Interesting Support threshold Sup_{min}, database *DB* and new dataset *db*. A complete collection of frequent item-sets may be derived by algorithm FIIU.

Proof. i: Complete collection of item-sets of *db* can be derived by using general frequent item-sets mining algorithm. ii: Because of THL and FITr storing all item-sets of DB, the new FITr and THL derived after running FIIU will include all frequent item-sets, according to i and Lemma 1.

5 FP-Tree Based Distributed Incremental Updating of Frequent Item-Sets

Experiment in section 6 will show that time cost of FIIU increases dramatically when the number of items in dataset increases. As all known, building distributed systems is a good method to improve performance. Because of FP-tree and *algorithm 2* being suitable for distributed architecture, we propose an algorithm DFIIU, based on FIIU and FP-tree, to incrementally mine frequent item-sets on multiple computers at the same time.

FP-growth algorithm mines frequent item-sets on FP-tree as follows. First, build a head item table. Then, construct the condition FP-tree of each item in head item table and perform mining recursively on such a tree. Motivated by FP-growth, it should be feasible to hand out works of constructing condition FP-tree of each item and mining frequent item-sets on such a tree to different computers in a distributed system.

Let NC_i be No.i computer in the distributed system. Assume that there are k NC in the distributed system. The distributed incremental mining of frequent item-set proceeds as follows. First, one NC, called server, builds FP-tree of new dataset *db* and divides items in the head item table of the FP-tree to k parts. Then the server sends the FP-tree and the divide to other NC. NC_i build conditional pattern trees of No.i part of items and mining frequent item-sets on such these trees. Item-sets derived from NC_i are stored in $WITs^i$. NC_i divides all linked-lists in WITs to k parts, WITs(1),…., WITs(k). After merging No.i part of linked-lists in other computers ($WITs^1$ (i), …, $WITs^k$ (i)), NC_i reads FITr stored on server's disk, and then searches frequent item-sets belonged to No.i part of linked-lists from FITr and inserts them to WITs(i). NC_i reads No.i part of linked-lists in THL, THL(i), which is stored in server's disk. After running algorithm GetFITr, The $FITr^i$ may be built. Finally, server merges frequent item tree form $FITr^1$ to $FITr^k$, and a new FITr is derived.

Algorithm DFIIU includes client algorithm DFIIU-C and server algorithm DFIIU-S.

Algorithm 4 DFIIT-C
Input: fp-tree, *item*, *class*
Symbol: Let $item_i$ be an array save No.i part of items, *class* be an array record which part of WITs each linked-list belongs to; there are *k* computers in distributed system.
Output: FITr
Method:
(1) *IS*=FP-Growth($item_i$, fp-tree);
(2) for each Item-set *A* in *IS*, InsertSI(*A*, $WITs^i$);
(3) Divide(*class*, $WITs^i$); // divide $WITs^i$ to k parts according *class*
(4) for(int j=0; j<*k*; j++) Send(NCj, $WITs^i$ (j)); //send No.j part $WITs^i$ to NCj
(5) for(int j=0; j<*k*; j++) Receive(NCj,$WITs^j$(i)); // receive No.i part WITs from NCj.
(6) WITs(i)=Merge($WITs^1$(i),…, $WITs^k$(i)); // merge all WITs(i)
(7) FITr=ReadFIT(); //read FITr from server's disk
(8) THL(i)=GetTHL(*class*); // get No.i part of linked-lists in THL from disk by *class*
(9) Return GetFITr (FITr, WITs(i) , THL(i));

Assume that algorithm DFIIU-C runs on NC_i, No.i client of distributed system. Function FP-Growth ($item_i$, fp-tree), in step 1, build conditional pattern trees of items in $item_i$. Then use FP-growth algorithm to mine all item-sets on such these trees and return collection of item-sets. Finally, use algorithm GetFITr to build a new FITr.

Algorithm 5 DFIIU-S
Input: db
Output: FITr
Method:
(1) FP-tree=CreateFPT(db); //build fp-tree of db
(2) item=GroupItem(fp-tree); // Save items' divide to array *item*
(3) class=GroupCls(WITs); // Save linked-lists' divide to array *class*
(4) for(int i=1;i<=k;)
(5) $FITr^i$=NC_i.DFIIU-C(FP-tree,item,class);//call Remote Method in other NC_i.
(6) FITr=Merge($FITr^i$,…,$FITr^k$); //merge all FITrs;

Lemma 3. Given Interesting Support threshold Sup_{min}, database *DB* and new dataset *db*. A complete collection of frequent item-sets may be derived by algorithm DFIIU.

Proof. The two-level hash index ensures that an item-set will be given a unique *class-name*. Therefore, same item-sets will be divided into same parts of WITs. We can conclude that there exists no item-set emerging in two computers' WITs in distributed system by step 3, 4, 5, 6 of algorithm 4. According to lemma 1 and lemma 2, a complete collection of frequent item-sets may be derived by *algorithm* DFIIU.

6 Experiment and Performance Analysis

6.1 Test Environment and Test Datasets

In this section, we evaluate FIIU and DFIIU in comparison with FP-growth. The three algorithms were implemented in java. Experiments were performed on a Intel C3 1.0G

PC with 512M main memory, running Windows 2000 sever and SQL Server 2000. There are two computers in the distributed system. Let *total* be the number of transactions, *d* be the number of new transactions. In the experiment, run time of FIIU and DFIIU were compared with run time of FP-growth when support threshold is Sup_{min} and the number of transactions is *total+d*. In the experiment, Sup_{min} is 0.1%.

We use synthetic dataset in the experiment because real datasets are not enough large for the experiment. In addition, we will illustrate that experimental conclusion has nothing to do with datasets in section 6.2.The synthetic datasets were generated from algorithm 6.

Algorithm 6 generating test datasets

Input: *Num* is the number of items in dataset, *Total* is the number of transactions; *Len* is length of longest transaction in dataset.

Output: dataset *D*

Method:

(1)*I*=CreateItems(*Num)*; //build collection of items
(2)For(int i=0;i<Total;i++){ //generate *Total* transactions
(3) T=Ø;
(4) len=Random(Num); //randomly generate a number between 1 to *Num*
(5) For(int j=0;j<len;j++){ //generate one transaction, *len* is length of transaction
(6) it=ChooseItem(*I*,T); //choice one item from *I*, which is not in T.
(7) InsertTrans(it,T)}; //put item to the transaction
(8) InsertDataset(T,D)}; //insert transaction into dataset

An integer, between 1 to *Num*, represents an item. Let *I* be collection of item, therefore, *I*={1,2,…,*Num*}. A transaction is a set of items in *I*, and there exists no repeat item in the transaction. The longest transaction is {1,2,…,*Num*}. *Len* is not greater than *Num* because there are no repeat items in a transaction.

6.2 Experimental Analysis

Assume that the number of items in *DB*+*db* is *Num* (it means that the longest transaction is {1,2,…, *Num*}). In the experiment, we compare time cost of FP-growth mining frequent item-sets on *DB*+*db* with time cost of FIIU while all item-sets in *DB* were stored in FITr and THL.

Time cost of FP-growth, as all known, will increase when size of *DB*+*db* increase.

Let time cost of FIIU be $t_{rf} + t_g + t_i$, where t_{rf} is time of reading FITr from disk, t_g is time of mining item-sets on *db*, and t_i is time of getting new FITr. There exists 2^{Num}-1 item-sets at the most when the number of items in *DB*+*db* is *Num*. Assume that there are 2^{Num}-1 item-sets in *DB*, and all the item-sets of *DB* are stored in FITr and THL, t_{rf} and t_i will not increase when size of *DB* increase. Let t_g be fixed, we can conclude that time cost of FIIU will not increase when size of DB increase.

By the above analysis, a characteristic can be derived that there exists a value *N*, the performance of FIIU will be better than FP-Growth (or other incremental updating algorithms that time cost increasing with size of dataset increasing) when size of DB is greater than *N*. Furthermore, it can be concluded that the characteristic has nothing to do with the chosen datasets. The *test 1, 2, 3* proved the characteristic.

DFIIU may further reduces the time cost because it hands out parts of mining work to others computer in distributed system although I/O cost counteract some of effects, which is proved in the *test 1, 2, 3*.

6.3 Experiment Results

In the test, we use N, L, T and d denotes *Num*, *Len*, *Total* and |*db*| respectively.

In *test 1*, as shown in Fig.4, N=15, L=10, d=5k. We report experimental results on 8 different sizes of dataset. It can be observed that performance of DFIIU and FIIU is better than FP-growth When T>200K and T>350K respectively, and average time of DFIIU is less than FIIU.

In *test 2*, as shown in Fig.5, N=17, L=17, d=10k. Experiments were performed on 5 different sizes of datasets. Performance of DFIIU and FIIU is better than FP-growth When T>700K and T>1200K respectively,

In *test 3*, as shown in Fig.6, N=19, L=19, d=10k, experiments were performed on 6 different size of datasets. Performance of DFIIU and FIIU is better than FP-growth When T>800K and T>1.3M respectively.

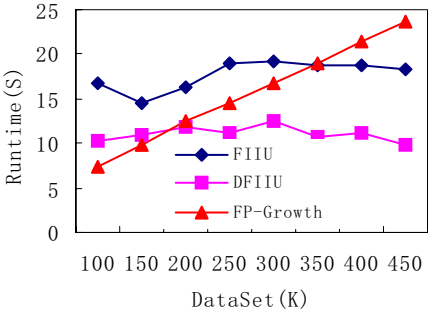

Fig. 4. N=15 L=10 d=5k

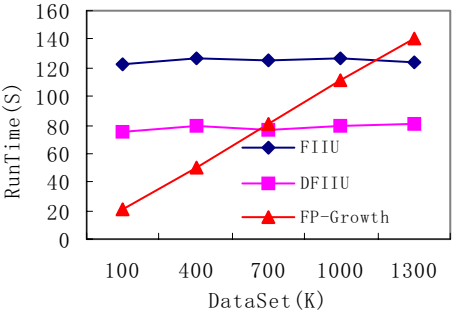

Fig. 5. N=17 L=17 d=10k

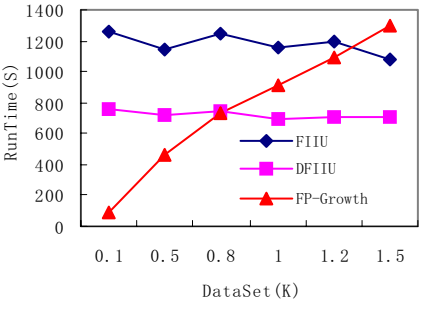

Fig. 6. N=19 L=19 d=10k

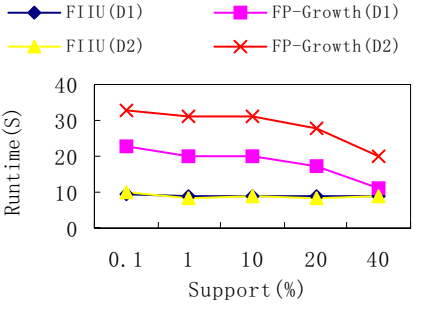

Fig.7. N=17 L=17

Fig.7 shows experiment result about first problem of incremental updating on two datasets. In first one, denotes as D1, N=17, L=17, T=100K. In D2, N=17, L=17,

T=200K. Time cost of FIIU was compared with FP-Growth on five different support thresholds. We can observe from Fig.7 that time cost of FIIU keep unchanged and that of FP-Growth increase when size of dataset increase. For each dataset, average time cost of FP-Growth is larger than FIIU on five thresholds. FIIU have obvious advantages on solving first problem of incremental updating than FP-Growth.

7 Conclusions

In this paper, we have proposed efficient method for three problems of incremental updating of frequent item-sets. Experiments also have proved that FIIU and DFIIU have better performance than FP-Growth (or other size sensitive incremental updating algorithms) when size of datasets is large. Especially, FIIU has a great advantage on first problem of incremental updating. However, we must indicate that finding all item-sets, which FIIU and DFIIU based on, is a NP hard problem. It may be an impossible task to mine all item-sets when the number of items on dataset is great. Therefore, for second and third problem of incremental problems, FIIU and DFIIU can show better performance when the number of items on dataset is less.

References

1. R.Agrawal,T.Imielinski,and A.Swami. Mining assciation rules between sets of items in large database[A]. the ACM SIGMOD. Washington, 1993.
2. R.Agrawal and R.Srikant.Fast algorithms for mining association rules[A].In:Proc. the 20 th International Conference on VLDB Santago,1994.487-499.
3. J.S.Park, et al. An efficient hash-based algorithm for mining association rules[A]. In Proc.1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'95), San Jose, CA.
4. A. Savasere, et al. An efficient algorithm for mining association rules in large databases [A]. In Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95), Zurich, Switzerland.
5. Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation [A]. In Proc. 2000 ACMSIGMOD Int. Conf. Management of Data (SIGMOD'00).
6. W. Cheung and O. R. Zaiane. Incremental Mining of Frequent Patterns Without Candidate Generation or Support Constraint[A]. In Proc. IDEAS 2003, Hong Kong.
7. Zhu Y.,Sun Z.,Ji X.. Incremental Updating Algorithm Based on Frequent Pattern Tree fof Mining Association Rules[J].Chinease Journal of Computers,2003,26(1):91-96.
8. Xiu-Li Ma,Yun-Hai Tong.Efficient Incremental Maintence of Frequent Patterns with FP-Tree[J].J.computer Science and Technology,2004,19(6),876-884.
9. R.Agrawal, and J.C.Shafer[A]. Parallel mining of association rules.IEEE Transaction on Knowledge and Data Engineering,1996,8(6),962-969.
10. E.H.Han, G.Karypis, and V.Kumar.Scalable parallel data minig for association rules[A].In Proc. ACM SIGMOD International Conference on Management of Data(SIGMOD'97).
11. Zhang Z.G. Study of Data Mining Algorithms on Massive Data[D].Harbin:Harbin Institute of Technology,2003.
12. S. Brin, R. Motwani, and C. Silverstein. Generalizing association rules to correlations[A]. In Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (ACM SIGMOD '97).