The ubiquitous data streams english language essay

Linguistics, English

AbstractWith pervasion of wireless networks and the wide-spread use of mobile devices, more and more streaming data are being generated anytime and anywhere. Hence, ubiquitous data stream mining has become an important research issue. As for frequent itemset mining, mining only closed frequent itemsets is a more efficient and memory-saving way which is critical in a resource-constrained circumstance. This paper deals with the issue of mining closed frequent itemsets from data streams with mobile devices. We propose a resource-aware closed fre-quent itemset mining algorithm, called CFP-Mine, over stream sliding windows. The information of closed item-sets is maintained in a CFP-Tree, which is a prefix-tree like, compact data structure that enables efficient mining. The structure of the CFP-Tree can be adjusted to further reduce memory usage. Because of its resource-aware fea-ture, the CFP-Mine algorithm can adapt to resource-constrained circumstances and utilize resources more ef-fectively than existing mining algorithms. Experiments show that CFP-Mine is efficient in terms of time and space and is applicable to mobile devices in reality. 1. IntroductionFrequent itemset mining is an important data mining problem with broad applications. A major challenge is that a huge number of frequent itemsets may be generated if there exist long frequent itemsets or the minimum sup-port is low. Hence, the concept of closed frequent itemset was introduced [6]. A frequent itemset is closed if there does not exist any proper super-itemset with the same support. The set of closed frequent itemsets can be used to derive the whole set of frequent itemsets. Yet, the number of closed frequent itemsets is usually much less than that of all frequent itemsets. Therefore, mining only closed frequent itemsets is more efficient and memory-saving.

With advances of the Internet, communications tech-nology, and sensor networks, many emerging applications generate data in the form of data streams, such as packets on the Internet, click streams, telecommunications data, sensor data, stock market data, etc. Because data streams are massive, fast, real-time, and unpredictable, traditional data mining techniques are not applicable. Hence, data stream mining has attracted much research attention from the data mining community. For some organizations and individuals, data will lose their value if not analyzed immediately. Recently, due to pervasion of wireless networks and increasing computing power of hand-held mobile devices such as smart phones and tablet computers, users are able to analyze data with mobile devices anytime and anywhere, which is called ubiquitous data mining. Ubiquitous data mining has many applications. For example, traveling investors can monitor stock portfolios from stock market data and traveling salespeople can perform customer profiling. Because of the constrained resources of mobile de-vices, however, existing data stream mining algorithms cannot be used for mobile devices. Insufficient resources will lead to mining interruption. Therefore, the concept of ubiquitous data stream mining has been proposed [2], which is to mine data streams with mobile devices and provide real-time and valuable information to the user. This paper deals with the issue of mining closed frequent itemsets from data streams with mobile devices. We pro-pose a resource-aware closed frequent itemset mining algorithm, called CFP-Mine (Closed Frequent Pattern Mining), over stream sliding windows. The information of closed itemsets is maintained in a CFP-Tree (Closed Frequent Pattern Tree), which is a prefix-tree like, com-pact data structure that enables efficient mining. The structure of the CFP-Tree can be adjusted to further re-duce memory usage. Because of its resource-aware fea-ture, the CFP-Mine algorithm can adapt to resource-constrained circumstances and utilize resources more ef-fectively than existing mining algorithms. Experiments show that CFP-Mine is efficient in terms of time and space and is applicable to mobile devices in reality. The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the CFP-Mine algorithm and related data structures. Section 4 shows experimental results. Section 5 concludes this paper. 2. Related workA lot of research is engaged in mining closed frequent itemsets in a static database. In [7], the CLOSET algo-rithm uses a compressed FP-tree-based structure to keep track of all closed itemsets without candidate generation for mining closed frequent itemsets. It develops a single prefix path compression technique to identify closed fre-quent itemsets quickly, and explores a partition-based projection mechanism for scalable mining in large data-bases. In [10], the CHARM algorithm represents datasets in a vertical format. It explores both itemset space and transaction space, adopts the Diffset technique to keep track of the mismatches in the tids of a candidate pattern from its parent pattern to get smaller tidsets, and elimi-nates non-closed itemsets, thus showing better perform-ance than CLOSET. In [8], the CLOSET+ algorithm com-pares the pros and cons of previously proposed mining algorithms CLOSET, CHARM, and OP, and integrates1their advantages. Using a depth-first search, a horizontal-format data structure, a hybrid-tree projection, and an item skipping technique to prune unpromising subsets, CLOSET+ outperforms the three previous mining algo-rithms. In [4], the FPclose algorithm uses multiple FP-tree-based data structures CFI-trees for mining closed frequent itemsets. It checks the closeness of frequent itemsets out of CFI-trees and employs array-based tech-niques along with various optimization methods. There is also some research on mining closed frequent itemsets in data streams. In [1], Chi et al. proposed an incremental mining algorithm, called Moment, with a compact data structure, called closed enumeration tree (CET), to update the most recent closed frequent itemsets over a data stream sliding window. CET is an in-memory data structure, which categorizes its nodes into four types and monitors the boundary of itemsets between closed frequent itemsets and the others. In [5], the CFI-Stream algorithm mines closed frequent itemsets over stream sliding windows. It only maintains a lexicographically ordered DIU Tree (Direct Update Tree), which stores closed frequent itemsets and their support information. In addition, when a new transaction arrives, it performs clo-sure checking � only associated closed itemsets and their support information are updated � and thus improving time efficiency. Experiments show that CFI-Stream achieves a better performance than Moment in terms of time and space. However, since it generates a lot of re-peated subsets, it would spend redundant time checking identical subsets more than once. In [9], the CloStream algorithm explores closed frequent itemsets over stream sliding windows by computing the intersection of a trans-action and all the existing closed itemsets related to it, and omits the process of searching from the structure that stores previous closed itemsets. In [3], Gaber and Yu proposed three types of resource-aware techniques: AIG (Algorithm Input Granularity), AOG (Algorithm Output Granularity), and APG (Algo-rithm

Processing Granularity). AIG reduces the input data rate by sampling, load shedding or creating data synopsis to adapt to limited bandwidth or power. AOG reduces the output data rate to adapt to limited memory. APG changes the support threshold of data stream mining algorithms to adapt to limited CPU processing speed. 3. CFP-Mine algorithmIn this section, we present CFP-Mine which is a re-source-aware algorithm for mining closed frequent item-sets over data stream sliding windows. The information of closed itemsets is maintained in a CFP-Tree, which is a prefix-tree like, compact data structure that enables effi-cient mining. When performing data mining on mobile devices, the limitation of memory is a critical factor that has to be considered. Therefore, by introducing an auxil-iary data structure ItemTable, the structure of CFP-Tree can be adjusted to further reduce memory usage. 3. 1. Data structuresCFP-Mine maintains two data structures: CFP-Tree and ItemTable. All closed frequent itemsets can be dis-covered efficiently from the CFP-Tree. The ItemTable is used to compress the CFP-Tree to reduce memory usage. 3. 1. 1. CFP-Tree. The CFP-Tree keeps all the closed itemsets in the current sliding window. A closed itemset is represented by a path from the root. The root is a NULL node. Except for the root node, each node in the tree contains the following fields:

?

item: the item name of this node

?

count: If the path from the root to this node represents a closed itemset, the count is the number of occur-rences of the closed itemset in the current sliding win-dow; otherwise, the count is 0. A node whose count is greater than 0 is referred to as a closed node. A node whose count is equal to 0 is referred to as an intermediate node.

?

child []: the pointer(s) to the child node(s) of this nodeIt is a hash table. From this node, we can find the child node whose item name is �a� by child [�a�].

?

parent: the pointer to the parent node of this node

?

next: the pointer to the next node that has the same item name as this nodeBesides tree nodes, the CFP-Tree includes the follow-ing two fields:

?

root: the pointer to the root node of the CFP-TreeThis field facilitates the access of any closed itemset.

?

nodeCnt: the current number of nodes in the CFP-TreeThis field can be looked up as a measurement of mem-ory usage. 3. 1. 2. ItemTable. The ItemTable comes in use when ad-justing the structure of the CFP-Tree. Each row of the ItemTable is associated with an item and contains three fields:

?

item: the item name

?

count: the number of occurrences of the item in the current sliding window

?

first: the pointer to the first node of the item in the CFP-TreeStarting from this pointer, we can find all nodes that represent the same item by following the �next� point-ers of these nodes. Using the hashing technique, we can locate the row as-sociated with a specific item �a� by itemTable [item = �a�]. The order of rows in the ItemTable determines a unique ordering of items for any transaction. We sort the2items of a transaction by this ordering before inserting it to the tree, so that the overlapping part of several itemsets can share the �prefix� of their paths in the CFP-Tree. Figure 1 shows the first sliding window of a data stream and the resulting data structures. The window size is set to 4 indicating a window consists of 4 transactions. Lemma 4. An itemset Z, where Z ? ?, Z ? X, Z is a sub-set of X, and Z is not closed in W1, becomes a closed itemset in W2 if and only if there exists a closed itemset Y in W1 such that X ? Y = Z. Let Sup1 (Z) and Sup2 (Z) be the support counts of Z in W1 and W2, respectively.�if� partFor any such Y, Sup1 (Y) ? Sup1 (Z). After the insertion of X, we have Sup2 (Z) = Sup1 (Z) + 1 and Sup2 (Y) = Sup1 (Y) for all Y.? Sup2 (Z) = Sup1 (Z) + 1 > Sup1 (Y) = Sup2 (Y)? The support count of Z in W2 is larger than any of its closed supersets.? For any superset S of Z, we can always find a closed superset Y of Z such that Sup2 (S) = Sup2 (Y) by the property of closed itemsets.? The support count of Z in W2 is larger than any of its supersets. current windowT1:{a, b, c}T2:{b, c, d}T3: {a, b, c, e}T4:{b, e, f}closed itemsets(support count){a, b, c} (2){b, c, d} (1){a, b, c, e} (1){b, e} (2){b} (4){b, e, f} (1){b, c} (3)Figure 1 The first sliding window and the resulting data structures? Z is closed in W2.�only if� partAssume there does not exist any Y such that X ? Y = Z. Case 1: Z is not a subset of any closed itemset Y in W1Only transaction X contributes to the support count of Z, so Sup2 (Z) = Sup2 (X). 3. 2. Determination of closed itemsetsSince X is a superset of Z, Z is not closed. Let W1 and W2 denote the sliding windows before and after a transaction X is inserted, respectively. Case 2: Z is a subset of some closed itemset Y in W1Since Z is not closed in W1, we can take a Y such that Sup1 (Z) = Sup1 (Y), Z ? Y, and X ? Y ? Z. Lemma 1. For any itemset Y which is not a subset of X, whether Y is closed or not remains the same after the in-sertion of a transaction X into the sliding window.? Z ? X ? Y and Sup2 (X ? Y) = Sup1 (X ? Y) + 1 = Sup2 (Z) = Sup1 (Z) + 1? Z is not closed. Since Y is not a subset of X, the support counts of Y and all supersets of Y will not change after the insertion of the transaction X into the sliding window. Conse-quently, whether or not Y is closed in W2 is the same as in W1. Based on the above lemmas, we can derive the follow-ing theorem on the set of closed itemsets in a sliding win-dow. Our algorithms are based on this theorem. Theorem 1. Let C (W) denote the set of closed itemsets in a sliding window W. C (W2) = C (W1) ? X ? {Z | ? Y ? C (W1) such that X ? Y = Z}Lemma 2. A closed itemset Y remains closed after the insertion of a transaction X into the sliding window. Case 1: Y is not a subset of XAccording to Lemma 1, Y remains closed after the in-sertion. 3. 3. Addition of a transactionCase 2: Y is a subset of XWe use a hash table waitingCIs to store the closed itemsets that will be used to update the CFP-Tree. It takes an itemset as the key and the support count of the itemset after a transaction X enters the current sliding window as the value. For example, waitingCIs [I] = c indicates that the support count of the closed itemset I after update is c. The support count of Y increases by 1 after the inser-tion of X. For any superset of Y, its support count either increases by 1 or remains unchanged. Therefore, Y re-mains closed after the insertion. Lemma 3. A transaction X becomes a closed itemset after being inserted into the sliding window. 3. 3. 1. Addition algorithm. When a transaction X enters the current sliding window, the CFP-Tree is updated by the following steps: After the insertion, the support count of any superset of X is less than that of X, so X is closed in W2. 31. Sort the transaction XWe sort X according to the item ordering in the Item-Table. Since all itemsets in the CFP-Tree follow this or-dering, sorting allows us to perform efficient intersection. In addition, the closed itemsets generated by intersection will also follow this ordering. 2. Put the transaction X into waitingCIsA transaction is a closed itemset after insertion. Thus, we set waitingCIs [X] = 1. 3. Intersect the transaction X with each of existing closed itemsets in the CFP-Tree and store generated closed itemsets and their counts into waitingCIsWe traverse the CFP-Tree to intersect X with each of existing closed itemsets. The structure of the CFP-Tree is very suitable for performing such intersection. The com-mon prefixes of several closed itemsets share the same path. While performing a depth-first traversal of the CFP-Tree, we can mark the items that belong to both the new transaction X and the itemset represented by the path from the root to the current node. When we encounter a closed node, we have finished intersecting the new trans-action X with an existing closed itemset. All marked items compose the intersection. Let Z be the intersection of the new transaction X and a closed itemset Y. Let d be the support count of Y in the CFP-Tree. If Z is not yet in waitingCIs, we set waitingCIs [Z] = d + 1. If Z already exists in waitingCIs, we compare its support count (waitingCIs [Z]) with d + 1. If d + 1 is larger, we update waitingCIs [Z] to d + 1. Repeat this process until all existing closed itemsets have been inter-sected with X. 4. Update the CFP-Tree using the closed itemsets stored in waitingCIsNow we have in waitingCIs all the closed itemsets that will be used to update the CFP-Tree. For each closed itemset Z in waitingCIs, update is done by going down a path that represents Z. Change the count of the last node in the path to waitingCIs [Z]. We create new nodes in the CFP-Tree in the case that the desired path does not exist. During the update process, we also maintain the �count� and �first� fields of the ItemTable. Create a new row if it is the first time an item appears. 3. 3. 2. Example. Starting with the example in Figure 1, a transaction T5 = {a, c, d, e} enters the current sliding win-dow. 1. Sort T5 according to the item ordering in the Item-Table. The result is {a, c, d, e}. 2. Put T5 into waitingCIs. The content of waitingCIs now is: IwaitingCIs [I]{a, c, d, e}13. Intersect T5 with each of existing closed itemsets and store generated closed itemsets and their counts into waitingCIs. The content of waitingCIs now is: IwaitingCIs [I](generated from which closed itemset){a, c, d, e}1transaction X{a, c}3{a, b, c}{c, d}2{b, c, d}{c}4{b, c}{a, c, e}2{a, b, c, e}{e}3{b, e}4. Update the CFP-Tree. Figure 2 shows the result after the insertion of T5. 3. 4. Deletion of a transactionWe use a hash table superCIs to store the itemsets that will be used to update the CFP-Tree. It takes an itemset as the key and a closed superset of the itemset as the value. 3. 4. 1. Deletion algorithm. When a transaction X leaves the current sliding window, the CFP-Tree is updated by the following steps: 1. Sort the transaction XWe sort X according to the item ordering in the Item-Table. The purpose, similar to that of the addition algo-rithm, is to make intersection more convenient and effi-cient. 2. Intersect the transaction X with each of existing closed itemsets in the CFP-Tree and store generated itemsets and theirs closed supersets into superCIscurrent windowT1:{a, b, c}T2:{b, c, d}T3:{a, b, c, e}T4: {b, e, f}T5: {a, c, d, e}closed itemsets(support count){a, b, c} (2){a, c, e} (2){b, e} (2){e} (3){a, b, c, e} (1){b} (4){b, e, f} (1){a, c} (3){b, c} (3){c} (4){a, c, d, e} (1){b, c, d} (1){c, d} (2)Figure 2 The data structures and the current sliding window after inserting transaction {a, c, d, e}43. 5. Adjustment of the CFP-TreeWe traverse the CFP-Tree to intersect X with each of existing closed itemsets. For each generated itemset Z, we find the corresponding closed itemset S, which is the su-perset of Z with the second highest count (the itemset with the highest count is Z itself) and is not a subset of X. superCIs takes Z as the key and S as the value. When the available memory of the mobile device is low, we can adjust the structure of the CFP-Tree to fur-ther reduce memory usage. A set is unordered. The two paths, a? b? c and b? c? a, represent the same itemset {a, b, c}. Therefore, we can adjust the structure of a prefix tree so that more nodes can be shared. 3. Update the CFP-Tree using the itemsets stored in su-perCIsSince Z is a subset of X, the count of Z decreases by 1 when X leaves the current sliding window. If Z is no longer a closed itemset, which happens when the count of Z equals to that of one of its supersets, we will delete Z from the CFP-Tree. 3. 5. 1. Adjustment algorithm. According to the �count�

field in the ItemTable, we can figure out which items ap-pear more frequently so that the nodes representing them should be closer to the root. In the adjustment algorithm, we sort the ItemTable using Bubble Sort and change the ordering of items in every closed itemset of the CFP-Tree by swapping nodes. The reason for using Bubble Sort is that it is simpler to exchange a parent node and a child node without losing itemset and count information. 3. 4. 2. Example. Continuing with the previous example, the transaction T1 = {a, b, c} leaves the current sliding window. 1. Sort T1 according to the item ordering in the Item-Table. The result is {a, b, c}. Here we describe the algorithm for swapping two nodes. Once we discover that the rows for items i1 and i2 should be exchanged, we can start from the �first� field of itemTable [item = i1] and follow the �next� field in each node to find all the nodes whose item name is i1. Let X be a node whose item name is i1. Let Y be the child node of X whose item name is i2. Let P be the parent of X. 2. Intersect T1 with each of existing closed itemsets and store generated itemsets and theirs closed supersets into superCIs. The content of superCIs now is: I (count)superCIs [I] (count){a, b, c} (2){a, b, c, e} (1){b, c} (3){b, c, d} (1){b} (4){b, e} (2){a, c} (3){a, c, e} (2){c} (4){c, d} (2)Swapping X and Y involves the following steps: 3. For each key in superCIs, its count decreases by 1. The itemset {a, b, c} is no longer closed since {a, b, c, e} has the same support count. The itemset {a, c} is also no longer closed since {a, c, e} has the same support count. Figure 3 shows the result after the deletion of T1. 1. Make a branchThere are two cases in which we have to �branch� the path P? X? Y.(a)X has child nodes other than YLet C be a child of X other than Y. The path root?�? P? X? C?� represents an itemset that does not contain Y. To maintain each of such paths, we create a new node X� and let it be a child of P. The value of the �item� and �count� fields of X� is the same as that of X. All child nodes of X are transplanted to X�.(b)X is a closed nodeThe path root?�? P? X represents a closed itemset that does not contain Y. To maintain this closed itemset, we create a new child node of P, X�, and assign the value of the �item� and �count� fields of X to it. Now X is no longer a closed node (count = 0) and the closed itemset is now represented by the path root?�? P? X�. 2. Exchange the positions of these two nodesThis is accomplished by modifying involving parent and child pointers. There is one case which we have to deal with carefully. If Y is a closed node, the path root?�? P? X? Y represents a closed itemset. After swapping X and Y, the path root?�? P? Y? X should represent the same closed itemset. Therefore, we should assign the count of this closed itemset to X and set the count of Y to 0. 3. Mark unnecessary nodescurrent windowT2: {b, c, d}T3: {a, b, c, e}T4: {b, e, f}T5: {a, c, d, e}closed itemsets(support count){a, b, c, e} (1){b, c} (2){c} (3){a, c, d, e} (1){b, c, d} (1){c, d} (2){a, c, e} (2){b, e} (2){e} (3){b} (3){b, e, f} (1)Figure 3 The data structures and the current sliding window after deleting transaction {a, b, c}5After swapping, Y becomes a child of P. P might have some other child whose item is the same as that of Y. In such a case, these two child nodes of P should be merged. After merging, we can remove the duplicate node by modifying involving parent and child pointers. However, it is difficult to modify the �next� pointer of involving nodes in order to achieve complete removal, since we cannot find the previous node of a node. Our solution is that, after all pairs of nodes that carry items i1 and i2 are swapped, we start from the �first� pointer of itemTable [item = i2] and follow the �next� pointers to �recycle� duplicate nodes. 3. 5. 2. Example. Figure 4 shows the CFP-Tree and Item-Table after adjustment from those in Figure 3. There are 16 nodes in Figure 3, and after adjustment the number of nodes is reduced to 13 in Figure 4. Adjusting the structure of the CFP-Tree makes it more compact and thus con-sumes less memory. Figure 4 CFP-Tree and ItemTable after adjustment3. 6. Discovery of closed frequent itemsetsWe can perform a depth-first traversal of the CFP-Tree to discover all closed frequent itemsets any time on de-mand. Because for any path in the CFP-Tree, the counts of nodes follow a descending order, we do not have to traverse the child nodes once the count of the current node is less than the user-specified minimum support. Therefore, the process of discovering closed frequent itemsets is very efficient. 4. Performance evaluationThrough empirical studies, we compare our algorithm with CFI-Stream [5], which is an efficient algorithm for mining closed frequent itemsets in data streams. We use Microsoft SQL Server 2005 with Microsoft Visual Studio Team Edition for Database Professional to generate syn-thetic datasets for our experiments. The three parameters of each synthetic dataset are the total number of transac-tions (T), the average length of transactions (L), and the number of different items (N), respectively. For example, the synthetic dataset T10k_L5_N1k contains 10000 trans-actions and 1000 different items, and each transaction consists of 5 items in average. Each transaction of a data-set is scanned only once in our experiments to simulate the environment of data streams. The minimum support is set to 50%. All experiments are conducted on a real mo-bile device, which is the hTC A7272 smart phone.

Figures 5 and 6 show the comparison of the average addition and deletion time, respectively, for CFP-Mine and CFI-Stream, where the horizontal axis is the window size (20, 40, 60, 80, and 100) and the vertical axis is the time in milliseconds (ms). The experimental data is the synthetic dataset T10k_L5_N1k. As the window size in-creases, the average addition time and deletion time in-crease as well but the average addition time of CFP-Mine increases just a little bit. The most significant advantage of CFP-Mine over CFI-Stream is in the average addition time, where CFP-Mine is thousands of times faster than CFI-Stream. CFP-Mine is slightly faster than CFI-Stream in the average deletion time. Because the execution of both CFP-Mine and CFI-Stream includes adding transac-tions, deleting transactions, and discovering closed fre-quent itemsets, overall CFP-Mine outperforms CFI-Stream in the execution time. Figure 5 Comparison of average addition timeFigure 6 Comparison of average deletion time6Figure 7 shows the comparison of the memory usage for CFP-Mine and CFI-Stream, where the horizontal axis is the different time and the vertical axis is the number of nodes in the tree structure. The experimental data is the synthetic dataset T100_L5_N50. Because it is difficult to get the precise memory usage of a particular application on a mobile device, we use the number of nodes in the tree structure of an algorithm to represent its memory usage. CFP-Mine uses less memory than CFI-Stream in average, and the CFP-Tree can be more compressed through adjustment to further reduce memory usage. 7Figure 7 Comparison of memory usage5. ConclusionIn this paper, we proposed the CFP-Mine algorithm for mining closed frequent itemsets from data streams with mobile devices. A prefix-tree like, compact tree structure CFP-Tree is used to maintain all the closed itemsets in the current sliding window. All the closed frequent itemsets in the current sliding window can be instantly discovered from the CFP-Tree when the user specifies a minimum support threshold. In addition, CFP-Mine is a resource-aware algorithm, which means it can adapt to resource-constrained circumstances and utilize resources more ef-ficiently than traditional mining algorithms. We compared CFP-Mine with CFI-Stream through empirical studies. Experiments show that CFP-Mine is efficient in terms of time and space and is applicable to mobile devices in real-ity.