Knowledge-Based Systems 49 (2013) 10–21

Contents lists available at SciVerse ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Efficient algorithms for frequent pattern mining in many-task computing environments

Kawuu W. Lin *, Yu-Chin Lo

Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, Kaohsiung, Taiwan

ARTICLE INFO

Article history:
Received 17 March 2012
Received in revised form 17 March 2013
Accepted 8 April 2013
Available online 28 April 2013

Keywords:
Data mining
Many-task computing
Cloud computing
Association rule mining
Frequent pattern mining

ABSTRACT

The goal of data mining is to discover hidden useful information in large databases. Mining frequent patterns from transaction databases is an important problem in data mining. As the database size increases, the computation time and required memory also increase. Because the number of items increases, the user behaviours also become more complex. To solve the problem of increasing complexity, many researchers have applied parallel and distributed computing techniques to the discovery of frequent patterns from large amounts of data. However, most studies have focused on improving the performance for a single task and have neglected the many-task computing issue, which is important in the current cloud-computing environments. In these environments, an application is often provided as a service, e.g., the Google search engine, implying that many users can use it simultaneously. In this paper, we propose a set of algorithms, containing the Equal Working Set (EWS) algorithm, the Request On Demand (ROD) algorithm, the Small Size Working Set (SSWS) algorithm and the Progressive Size Working Set (PSWS) algorithm, for frequent pattern mining that provides a fast and scalable mining service in many-task computing environments. Through empirical evaluations in various simulation conditions, the proposed algorithms are shown to deliver excellent performance with respect to scalability and execution time.

1. Introduction

Frequent pattern mining involves searching in a database for a pattern that appears more frequently than a specified threshold. An association rule is defined as $X \Rightarrow Y$, where X and Y are itemsets. Association rule mining is used to discover sets of items, i.e., frequent patterns, associated with other items in the database. Studies on frequent pattern mining are usually classified into two types: (1) the generate-and-test [3] (Apriori-like) approach and (2) the frequent pattern growth [9] approach (FP-growth-like). The Apriori-like methods iteratively generate a candidate itemset with size $(k+1)$ from frequent itemsets of size k and scan the database repetitively to test the frequency of each candidate itemset. The Apriori-like methods inherently suffer from the following two costs [9]:

- The cost in memory required to handle a large number of candidate itemsets is high. For example, if there are n frequent length-1 itemsets, there will be $n*(n-1)/2$ length-2 candidates. The method accumulates the frequency for each candidate, encountering two critical problems. The first problem is that the main memory is limited and cannot store whole candidates. The second problem is that even if the whole candidates can be loaded into the main memory, the performance of accumulating the frequency will be low because the method spends a large amount of computational time searching for the candidate in a large candidate pool, increasing the frequency for the candidate each time it is found. Moreover, to discover a length l frequent pattern, the method must generate 2^l-2 candidates. The scalability of Apriori-like methods is thus restricted by the high memory cost.
- The database scanning cost to discover frequent patterns is also high. Assume the length of the longest pattern in the database is l. The Apriori-like methods require l physical database scans for pattern discovery. A physical database scan consumes a large amount of computational time because of the resulting data explosion. To reduce the times of the database scans is one way to reduce execution time.

Han et al. [9] proposed a novel data structure, called frequent pattern tree (FP-tree), in which transactions are compressed and stored. A mining algorithm, FP-growth, was then proposed to discover frequent patterns from the FP-tree without generating candidates, therefore enhancing the scalability of the method. FP-growth requires only two scans of the physical database and also greatly reduces the execution time. After two scans of the database, the method constructs a header table and a FP-tree

* Corresponding author.
E-mail address: linwc@kuas.edu.tw (K.W. Lin).

0950-7051/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.knosys.2013.04.004

corresponding to the database. It then uses the FP-growth algorithm to discover frequent patterns by recursively reconstructing conditional FP-trees from the original FP-tree. An entry of the header table records an item (usually an ID) and the first appearance node in the FP-tree. A node of the FP-tree consists of an attribute for storing the item, a reference for its parent, a set of references for its children nodes, and a reference for the next node with the same item in the FP-tree. The scalability of FP-growth/FP-growth-like methods is limited by the main memory size because all of the reconstructed FP-trees with the header tables are stored in the main memory. Moreover, the scalability and the execution performance of FP-growth-like methods are restricted to the FP-tree size. In an FP-tree study, a large FP-tree is a tree that has many branches or a tree with a high number of average fan-out nodes. A large FP-tree is derived from one or a combination of the following three causes: data characteristics, such as a large number of items; user behaviour characteristics, such as a long transaction length; and mining parameters, such as a small support threshold. If an FP-tree is large, the methods need a large amount of memory to store the FP-tree, the conditional FP-trees and the header tables, as well as a large amount of computational time to reconstruct the conditional FP-trees to complete the mining. A large database might contain a large number of items, complex user behaviours, etc., causing a sharp increase in the computation time and required memory for mining frequent patterns.

Many studies on frequent pattern mining have proposed improvements in execution time efficiency. Parallel and distributed computing techniques have attracted attention for their ability to manage and compute large amounts of data [18]. The difficulties encountered when mining large databases launched research into the design of parallel and distributed algorithms [4,8] and architectures like grid [1,6,7,19,21,23], cluster [7,11,23,24], cloud [10,14,15,22]and shared nothing parallel machines [12] to solve the problem. Previous studies have also used multiprocessor architectures to improve performance [13], but the expense of the machines slowed down the advancement of this approach. In [5], the authors proposed a tree projection technique to achieve parallelism. The technique requires a large amount of memory and is therefore not very practical. The approach used in most current studies is to divide the database and distribute each section to nodes or processors for mining, thus distributing the computational load. During the mining process, the nodes exchange required transactions with one another. The data transmission via the network involves at least four layers, including the physical/data link layer, the network layer, the transport layer, and the combined session/presentation/application layer. Each layer uses its own protocol to pack the data in the sender and to unpack the data in the receiver. In addition to the protocol cost, transmission via the physical network is time-consuming. For this reason, the workload of exchanging data among nodes increases with large database size. Reducing the amount of data transmitted over the network can significantly improve the execution time. Although many algorithms have been proposed, the execution efficiency of frequent pattern mining remains a challenge to researchers because of the explosion of data.

In [16], the authors proposed the data mining method CARM, which efficiently utilises cloud nodes to discover frequent patterns in cloud computing environments. In these environments, workload balancing among computing nodes is one of the most critical issues affecting execution performance. CARM has demonstrated its superior workload balancing as compared with the well-known BTP-tree algorithm [25]; however, CARM focuses only on accelerating the mining process for a single task. Many-task computing is becoming popular, and should be considered. Many-task computing is used to bridge the gap between high throughput computing and high performance computing [17]. There are many applications where many-task computing would be useful, such as astronomy, bioinformatics, cognitive neuroscience, and data mining. A common characteristic of all these applications is that they are often provided as services, e.g., the Google search engine, implying that many users can use it simultaneously [17]. Therefore, an appropriate design for such environments is necessary.

The primary contributions of this study are: (1) a brief data structure to store tree reconstruction information and minimise data transmission on the network, (2) a set of algorithms based on CARM that discovers frequent patterns capable of providing fast and scalable service in many-task computing environments, and (3) a series of experiments to evaluate the performance of the proposed algorithms. Through empirical evaluations at various simulation conditions, we observed the proposed algorithm requires only 12.2% and 18% execution time, using TPFP-tree [24] and BTP-tree, respectively, when there are 100 mining tasks. The improvement increases with an increase in the number of mining tasks.

In the following sections, we briefly review related work in Section 2. Section 3 presents the algorithms for many-task frequent pattern mining. Section 4 provides an empirical evaluation of the performance of the algorithms. Section 5 presents the conclusions and future work.

2. Related work

In this section, we review previous studies on three subjects closely related to this research: (1) FP-tree and FP-growth, (2) parallel and distributed frequent pattern mining and (3) CARM algorithm.

2.1. FP-tree and FP-growth

Han et al. [9] proposed a tree-based data structure, *FP-tree*, and the corresponding mining algorithm, *FP-growth*, for discovering frequent patterns. The algorithm requires two database scans to complete the mining task. The first scan calculates the support for each item. This scan also creates a header table, recording the item name, its corresponding support and the first node-link linking to the first node in the FP-tree with the same item name. The support sorts items in the header table in descending order. For each transaction, items with support values under the threshold are filtered out in the second scan, and the remaining items are sorted in descending order using their support values. The sorted items in each transaction are inserted into the FP-tree. The FP-tree structure contains a root node labelled as null, a set of item-prefix subtrees as the children of root, and a header table. The FP-tree node structure is <item-name, count (support), node-link>, in which item-name is the item name used for identification, count is the number of transactions reaching this node by the same path from root, and node-link is a pointer linking to the next node in the FP-tree with the same item name.

To insert transaction P into FP-tree T, we check whether T has a child n such that *n.item-name* is identical to the item-name of the first element of P. If the node exists, the count of n is increased by 1. Otherwise, it creates a new node, m, with the same item name as n. The count of m is set to 1, the parent link is set to T, and m's node-link is set to the nodes with the same item-name using the node-link structure. We recursively perform the insertion for each item in P until each item is inserted into the FP-tree. After constructing the FP-tree, FP-growth is used to discover the frequent patterns. An item in the header table is selected to construct the conditional FP-tree by inserting all prefix paths of the item, which can be retrieved using the node-link structure in header table. The item name is called the conditional pattern base. The FP-growth is executed