Mining Frequent Itemsets in Distributed and Dynamic Databases

M. E. Otey C. Wang S. Parthasarathy
Computer and Information Science Dept.
The Ohio-State University
{otey, wachao, srini}@cis.ohio-state.edu

A. Veloso W. Meira Jr.
Computer Science Dept.
Universidade Federal de Minas Gerais
{adrianov, meira}@dcc.ufmg.br

Abstract

Traditional methods for frequent itemset mining typically assume that data is centralized and static. Such methods impose excessive communication overhead when data is distributed, and they waste computational resources when data is dynamic. In this paper we present what we believe to be the first unified approach that overcomes these assumptions. Our approach makes use of parallel and incremental techniques to generate frequent itemsets in the presence of data updates without examining the entire database, and imposes minimal communication overhead when mining distributed databases. Further, our approach is able to generate both local and global frequent itemsets. This ability permits our approach to identify high-contrast frequent itemsets, which allows one to examine how the data is skewed over different sites.

1 Introduction

Advances in computing and networking technologies have resulted in distributed and dynamic sources of data. A classic example of such a scenario is found in the warehouses of large national and multinational corporations. Such warehouses are often composed of disjoint databases located at different sites. Each database is continuously updated with new data as transactions occur. The update rate and ancillary properties may be unique to a given site.

A user may be interested in generating a global model of the database, thus the sites must exchange some information about their local models. However, the information exchange must be made in a way that minimizes the communication overhead. The frequency at which the global model is updated may vary from the frequency at which each local model is updated. Furthermore, in such a distributed scenario, the user may be interested in not only knowing the global model of the database, but also the differences (or contrasts) between the local models.

Analyzing these distributed and dynamic databases requires approaches that make proper use of the distributed resources, minimize communication requirements and reduce work replication. In this paper we present an efficient frequent itemset mining approach when data is both distributed and dynamic. The main contributions of our paper are:

1. A parallel algorithm based on the ZIGZAG incremental approach, which is used to update the local model;
2. A distributed mining algorithm that minimizes the communication costs for mining over a wide area network, which is used to update the global model;
3. Novel interactive extensions for computing high contrast frequent itemsets;
4. Experimentation and validation on real databases.

Related Work. In [1] three distributed approaches were proposed. COUNT DISTRIBUTION is a simple distributed implementation of APRIORI. DATA DISTRIBUTION generates disjoint candidate sets on each site. CANDIDATE DISTRIBUTION partitions the candidates during each iteration, so that each site can generate disjoint candidates independently of the other sites. [9] presents another distributed algorithm, PARECLAT, which is based on the concept of *equivalence classes*. These techniques are devised to scale up a given algorithm (e.g., APRIORI, ECLAT, etc.). They perform excessive communication operations and are not efficient when data is geographically distributed.

Some effort has also been devoted to incrementally mining frequent itemsets [5, 7, 8, 2, 4] in dynamic databases. The DELI algorithm was proposed in [5]. It uses statistical sampling methods to determine when the current model is outdated. A similar approach [4] monitors changes in the data stream. An efficient algorithm called ULI [7], strives to reduce I/O requirements for updating the frequent itemsets by maintaining the previous frequent itemsets and the *negative border* along with their supports.

The rest of the paper is organized as follows. In Section 2 we describe the algorithms and novel interactive extensions. In Section 3 we validate the algorithms through extensive evaluation. Concluding remarks are made in Section 4.

2 Parallel, Distributed Incremental Mining

In this section we describe our parallel and distributed incremental mining algorithms. Specifically, the idea is that within a local domain one can resort to parallel mining approaches (either within a node or within a cluster) and across domains one can resort to distributed mining approaches (across clusters). The key distinction between the two scenarios is the cost of communication.

Problem Definition. Let $\mathcal{I}$ be a set of distinct items. Let $\mathcal{D}$ be a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. A set of exactly k items is called a *k-itemset*. The *tidset* of an itemset C corresponds to the set of all transaction identifiers (*tids*) where the itemset C occurs. The *support* of C, is the number of transactions of $\mathcal{D}$ in which it occurs as a subset. The itemsets that meet a user-specified *minimum support* are referred to as *frequent* itemsets. A frequent itemset is *maximal* if it is not subset of any other frequent itemset.

Using $\mathcal{D}$ as a starting point, a set of new transactions d^+ is added forming Δ ($\mathcal{D} \cup d^+$). Let $s_\mathcal{D}$ be the minimum support used when mining $\mathcal{D}$, and F_D be the set of frequent itemsets obtained. Let Π be the information kept from the current mining that will be used in the next operation (in our case, Π consists of $F_\mathcal{D}$). An itemset C is frequent in Δ if its support is no less than s_Δ. If a frequent itemset in $\mathcal{D}$ remains frequent in Δ it is called a *retained* itemset.

The database Δ can be divided into n partitions, $\delta_1, ..., \delta_n$. Each partition δ_i is assigned to a site S_i. Let $C.sup$ and $C.sup_i$ be the respective support of C in Δ (global support) and δ_i (local support). Given s_Δ, C is *global frequent* if $C.sup \geq s_\Delta \times \mid \Delta \mid$; correspondingly, C is *local frequent* at δ_i, if $C.sup_i \geq s_\Delta \times \mid \delta_i \mid$. The set of all maximal global frequent itemsets is denoted as MFI$_\Delta$, and the set of maximal local frequent itemsets at δ_i is denoted as MFI$_{\delta_i}$. The task of mining frequent itemsets in distributed and dynamic databases is to find F_Δ.

The ZIGZAG Incremental Algorithm. The main idea behind the algorithm is to incrementally compute MFI$_\Delta$ using Π. This avoids the generation of many unnecessary candidates. Having MFI$_\Delta$ is sufficient to know which itemsets are frequent; their exact support are then obtained by examining d^+ and using Π, or, where this is not possible, by examining Δ. ZIGZAG employs a backtracking search to find MFI$_\Delta$ which is explained in [8].

The support computation is based on the associativity of itemsets – let $\mathcal{L}(\mathcal{C}_k)$ be the tidset of $\mathcal{C}_k$. The support of any itemset is obtained by intersecting the tidsets of its subsets. To avoid replicating work done before, ZIGZAG first verifies if the candidate is a retained itemset. If so, its support can be computed by using d^+ and Π.

Parallel Search for the MFI$_\Delta$. The main idea of our parallel approach is to assign distinct backtrack trees to distinct processors. Note that there is no dependence among the processors, because each tree corresponds to a disjoint set of candidates. Since each processor can proceed independently there is no synchronization while searching for MFI$_\Delta$. To achieve a suitable level of load-balancing, the trees are assigned to the processors by using the scheme of *bitonic partitioning* [3]. The bitonic scheme is a greedy algorithm, which first sort all the w_i (the work load due to tree i). w_i is calculated based on our ideas for estimating the number candidates using correlation measures, presented in [8]. Next it extracts the tree with maximum w_i, and assign it to processor 0. The next highest workload tree is assigned to processor 1 and so on.

2.1 Distributed Algorithm

Lemma 1 – **A global frequent itemset must be local frequent in at least one partition.** ■

Lemma 2 – $\bigcup_{i=1}^{n}$ **MFI$_{\delta_i}$ determines F_Δ.** ■

First, each site S_i independently performs a parallel and incremental search for MFI$_{\delta_i}$, using ZIGZAG on its database δ_i. After all sites finish their searches, the result will be the set {MFI$_{\delta_1}$, MFI$_{\delta_2}$, ... , MFI$_{\delta_i}$}. This information is sufficient for determining all global frequent itemsets. Next, the algorithm starts after all local MFIs were found. Each site sends its local MFI to the other sites, and then they join all local MFIs. Now each site knows the set $\bigcup_{i=1}^{n}$ MFI$_{\delta_i}$, which is an upper bound for MFI$_\Delta$. In the next step each site independently performs a top down incremental enumeration of the potentially global frequent itemsets, as follows. Each itemset present in the upper bound $\bigcup_{i=1}^{n}$ MFI$_{\delta_i}$ is broken into k subsets of size $(k-1)$. This process iterates generating smaller subsets and incrementally computing their supports until there are no more subsets to be checked. At the end of this step, each site will have the same set of potentially global frequent itemsets.

The final step makes a reduction on the local supports of each itemset, to verify which of them are globally frequent. The process starts with site S_1, which sends the supports of its itemsets to S_2. S_2 sums the support of each itemset with the value of the same itemset obtained from S_1, and sends the result to S_3. This procedure continues until site S_n has the global supports of all potentially global frequent itemsets. Then site S_n finds all itemsets that have support greater than or equal to s_Δ, which constitutes F_Δ (by Lemma 2).

2.2 Interactive Issues

High-Contrast Frequent Itemsets. An important issue when mining Δ is to understand the differences between $\delta_1, \delta_2, ..., \delta_n$. *An effective way to understand such differences is to find the high-contrast frequent itemsets.* The supports of such itemsets vary significantly across the databases. We use the well-established notion of entropy to detect how the support of a given itemset is distributed across the databases. For a random variable, the entropy is a measure of the non-uniformity of its probability distribution. Given an itemset $\mathcal{C}$, the value $p_{\mathcal{C}}(i) = \frac{\mathcal{C}.sup_i}{\mathcal{C}.sup}$ is the probability of occurrence of $\mathcal{C}$ in δ_i. $\sum_{i=1}^{n} p_{\mathcal{C}}(i) = 1$, and $H(\mathcal{C}) = -\sum_{i=1}^{n}(p_{\mathcal{C}}(i) \times log(p_{\mathcal{C}}(i)))$ is a measure of how the local supports of $\mathcal{C}$ is distributed across the databases. Note that $0 \leq H(\mathcal{C}) \leq log(n)$, and so $0 \leq E(\mathcal{C}) = \frac{log(n)-H(\mathcal{C})}{log(n)} \leq 1$. If $E(\mathcal{C})$ is greater than or equal to a given minimum entropy threshold, then $\mathcal{C}$ is classified as high-contrast frequent itemset.

Query Response Time. One of the goals of the algorithm is to minimize response time to a query for F_Δ in a dynamic, distributed database. Since Δ is dynamic, each site is incrementally updating its local frequent itemsets. The time it takes to update the local frequent itemsets is proportional to $\mid d^+ \mid$. We can view the updates to the database as a queue containing zero or more such blocks. If a query arrives while a block is being processed, there cannot be a response until the calculation of the local frequent itemsets is completed. An obvious approach to reducing response time is to decrease the size of d^+. However, because of overhead, the time it takes to do two increments of size $\mid d^+ \mid$ is longer than the time it takes to do a single increment of size $2 \times \mid d^+ \mid (n > 1)$. So there is a trade-off: *The larger d^+ is, the more up-to-date F_Δ will be, since it incorporates a greater number of changes to Δ, but the longer the response time to the query will be*.

3 Experimental Evaluation

Our experimental evaluation was carried out on two clusters. The first cluster consists of dual PENTIUM III 1GHz nodes with 1GB of main memory. The second cluster consists of single PENTIUM III 933 MHz nodes with 512 MB of main memory. We assume that each database is distributed between the clusters, and that each node in the cluster has access to its cluster's portion of the database. Within each cluster, we have implemented the parallel program using the MPI message-passing library (MPICH), and for communication between clusters we have used sockets. We used a real database for testing the performance of our algorithm. The WCup database is generated from the clickstream log of the 1998 World Cup web site. This database is 645 MB in size and contains 7,618,927 transactions. Additional experiments on other real and synthetic databases can be found in [6].

Intra-Cluster Evaluation. Figure 1(a) shows the execution times obtained for the WCup dataset in parallel and incremental configurations. As we can see, better execution

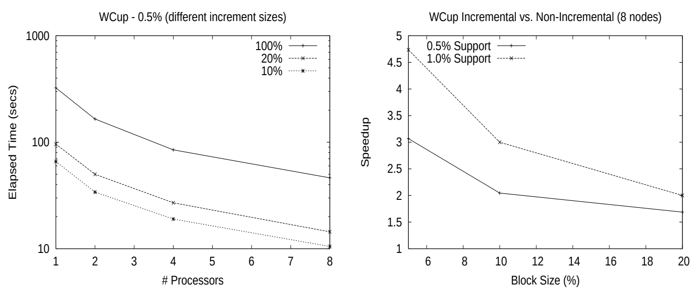

Figure 1. (a) Total Execution Times, and (b) Speedups on Different Incremental Configurations.

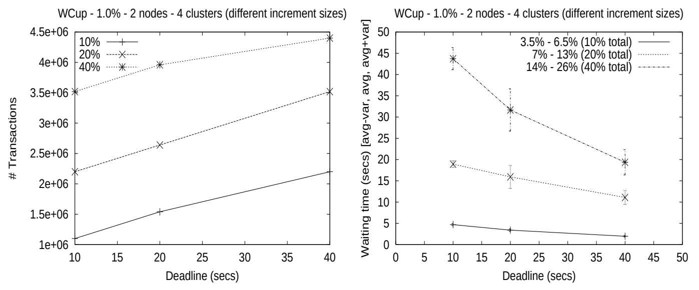

Figure 2. (a) Number of Transactions Processed, and (b) Query response time.

times are obtained when we combine both parallel and incremental approaches. Furthermore, for the same parallel configuration, the execution time is better for smaller block sizes, as Figure 1(b) shows.

Inter-Cluster Evaluation. The next experiment examines how the size of a block and the time at which a query arrives (the deadline) affects the amount of data used to build F_Δ. The lines on the graph represent different block sizes, which are given here as percentages of the database on each cluster. Figure 2(a) shows that the more time that elapses before the query arrives, the more data that is incorporated into the model, which is to be expected. It also shows that as the block size decreases, fewer transactions can be processed before the query arrives. *This is because there is more overhead involved in processing a large number of small blocks than there is in processing a small number of large blocks.* We also investigated the performance of our algorithm in experiments for evaluating the query response time (the amount of time a user must wait before F_Δ is computed). Figure 1(a) shows that as the increment size decreases, the time to wait for a response also decreases. However, the time at which a query arrives affects the waiting time in a somewhat random manner. This is because

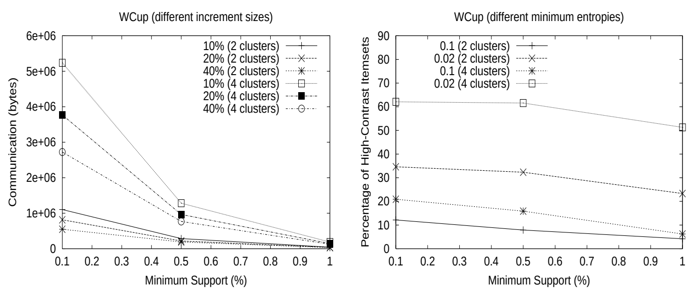

Figure 3. (a) Communication Overhead, and (b) High-Contrast Frequent Itemsets.

a query can arrive at any point during the processing of a block. The time remaining to compute the local frequent itemsets therefore varies. Figure 2(b) shows the query response time averaged over five runs, and the vertical bars represent the variance in the runs. The trade-off between block size and query response time is apparent: as the block size increases, the number of transactions processed also increases, but the response time increases as well. This shows that we can assign larger blocks to less powerful clusters, the basic idea being that the local model can be updated less frequently, but in response, smaller query response times can be obtained.

We also performed a set of experiments to analyze the communication overhead imposed by our algorithm. We examined the number of bytes transferred between clusters when we varied the minimum support, the block size, and the number of clusters involved in the computation. The results can be seen in figure 3(a). As is expected, as the minimum support decreases, the number of candidates will increase, and will therefore increase the number of bytes that must be transferred between the clusters, since our algorithm must exchange the supports of every candidate processed. Also, as the block size increases, the amount of communication decreases. The reason is that for smaller block sizes the number of candidates processed tends to be greater. Finally, the amount of communication required increases when more clusters are involved in the process.

The last set of experiments concerns high-contrast frequent itemsets. We varied three parameters: the minimum support, the number of clusters involved in the process, and the minimum entropy. The results are showed in Figure 3(b). As we can observe, very different results were obtained from each database. The percentage of high-contrast frequent itemsets is interesting here because *it reveals the skewness of the database*. Usually the percentage of high-contrast frequent itemsets will diminish as the minimum support threshold increases. This is quite understandable considering that when the support threshold is low, there will be a large number of global frequent itemsets generated, and many of these itemsets become global frequent only because they are frequent in some small number of sites, resulting in many high-contrast frequent itemsets. In contrast, as the support threshold increases, it becomes harder for a local frequent itemset to become global frequent, which results in a smaller number of high-contrast frequent itemsets. Meanwhile, the more clusters on which the data distributed, the greater the possibility of skewness in the data.

4 Conclusions

In this paper we considered the problem of mining frequent itemsets in dynamic and distributed databases. We presented an efficient distributed and parallel incremental algorithm to deal with this problem. Our experiments examined the trade-offs involved in minimizing the query response time (whether to sacrifice query response time in order to incorporate more transactions in the model) and the amount of data transferred between clusters. Future work involves using sampling techniques to minimize the query response time and investigating how to minimize query response time in wide-area networks, where communication latencies tend to be relatively large.

References

[1] R. Agrawal and J. Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 962–969, 1996.

[2] D. Cheung, J. Han, and et al. Maintenance of discovered associations in large databases: An incremental updating technique. In *Proc. of the Int'l. Conf. on Data Engineering*, 1996.

[3] M. Cierniak, M. Zaki, and W. Li. Compile-time scheduling algorithms for a heterogeneous network of workstations. In *The Computer Journal*, volume 40, pages 356–372.

[4] V. Ganti and J. G. et al. Demon: Mining and monitoring evolving data. In *Proc. of the 16th Int'l Conf. on Data Engineering*, pages 439–448, San Diego, USA, 2000.

[5] S. Lee and D. Cheung. Maintenance of discovered associations: When to update? In *Research Issues on Data Mining and Knowledge Discovery*, 1997.

[6] M. E. Otey, A. Veloso, C. Wang, S. Parthasarathy, and W. Meira. Mining frequent itemsets in distributed and dynamic databases. In *OSU-CISRC-9/03-TR48*, 2003.

[7] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules. In *Proc. of the 3rd Int'l Conf. on Knowledge Discovery and Data Mining*, August 1997.

[8] A. Veloso and W. M. J. et al. Mining frequent itemsets in evolving databases. In *Proc. of the Int'l Conf. on Data Mining*, Arlington, USA, 2002.

[9] M. Zaki and S. P. et al. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 4(1):343–373, 1997.