Parallel and Distributed Frequent Itemset Mining on Dynamic Datasets[⋆]

Adriano Veloso[1,2], Matthew Erick Otey[2]
Srinivasan Parthasarathy[2], and Wagner Meira Jr.[1]

[1] Computer Science Department, Universidade Federal de Minas Gerais, Brazil
{adrianov,meira}@dcc.ufmg.br
[2] Department of Computer and Information Science, The Ohio-State University, USA
{otey, srini}@cis.ohio-state.edu

Abstract Traditional methods for data mining typically make the assumption that data is centralized and static. This assumption is no longer tenable. Such methods waste computational and I/O resources when data is dynamic, and they impose excessive communication overhead when data is distributed. As a result, the knowledge discovery process is harmed by slow response times. Efficient implementation of incremental data mining ideas in distributed computing environments is thus becoming crucial for ensuring scalability and facilitate knowledge discovery when data is dynamic and distributed. In this paper we address this issue in the context of frequent itemset mining, an important data mining task. Frequent itemsets are most often used to generate correlations and association rules, but more recently they have also been used in such far-reaching domains as bio-informatics and e-commerce applications. We first present an efficient algorithm which dynamically maintains the required information even in the presence of data updates without examining the entire dataset. We then show how to parallelize the incremental algorithm, so that it can asynchronously mine frequent itemsets. Further, we also propose a distributed algorithm, which imposes low communication overhead for mining distributed datasets. Several experiments confirm that our algorithm results in excellent execution time improvements.

1 Introduction

The field of knowledge discovery and data mining (KDD), spurred by advances in data collection technology, is concerned with the process of deriving interesting and useful patterns from large datasets. Frequent itemset mining is a core data mining task. Its statement is very simple: to find the set of all subsets of items that frequently occur together in database transactions. Although the frequent itemset mining task has a simple statement, it is CPU and I/O intensive, mostly because the large number of itemsets that are typically generated and the large size of the datasets involved in the process.

Now consider the problem of mining frequent itemsets on a dynamic dataset, like those found in e-commerce and web-based domains. The datasets in such domains are constantly updated with fresh data. Let us assume that at some point in time we have computed all frequent itemsets for such a dataset. Now, if the dataset is updated, then

[⋆] This work was done while the first author was visiting the Ohio-State University the set of frequent itemsets that we had previously computed would no longer be valid. A naive approach to compute the new set of frequent itemsets would be to re-execute a traditional algorithm on the updated dataset, but this process is not effi cient since it is memoryless and ignores previously discovered knowledge, essentially replicating work that has already been done, and possibly resulting in an explosion in the computational and I/O resources required. To address this problem, several researches have proposed incremental algorithms [15,6,9,14,4], which essentially re-use previously mined information and try to combine this information with the fresh data to effi ciently re-compute the new set of frequent itemsets. The problem now is that the size and rate at which the dataset can be updated are so large that existing incremental algorithms are ineffective. Therefore, to mine such large and high-velocity datasets, we must also rely on high-performance multi-processing computing.

Two dominant approaches for using multiple processors have emerged: distributed memory (where each processor has a private memory), and shared memory (where all processors access common memory). The performance-optimization objectives for distributed memory approaches are different from those of shared memory approaches. In the distributed memory paradigm synchronization is implicit in communication, so the goal becomes communication optimization. In the shared memory paradigm, synchronization stems from locks and barriers, and the goal is to minimize these points. However, the majority of the parallel mining algorithms [1,3,5] suffer from high communication and synchronization overhead. In this paper we propose an effi cient parallel and incremental algorithm for mining frequent itemsets on dynamic datasets. Our algorithm makes use of both shared and distributed memory advantages, and it is extremely effi cient in terms of communication and synchronization overhead. Extensive experimental evaluation confi rm that it results in excellent execution time improvements.

1.1 Problem Definition

The frequent itemset mining task can be stated as follows: Let $\mathcal{I}$ be a set of distinct attributes, also called items. Let $\mathcal{D}$ be a set of transactions, where each transaction has a unique identifi er (*tid*) and contains a set of items. A set of items is called an *itemset*. An itemset with exactly k items (where k is a nonnegative integer) is called a k-itemset. The *tidset* of an itemset C corresponds to the set of all transaction identifi ers (*tids*) in which the itemset C occurs. The *support count* of C, is the number of transactions of $\mathcal{D}$ in which it occurs as a subset. Similarly, the *support* of C, denoted by $\sigma(C)$, is the percentage of transactions of $\mathcal{D}$ in which it occurs as a subset. The itemsets that meet a user specifi ed *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 the dynamic dataset Δ (i.e., $\Delta = \mathcal{D} \cup d^+$). Let $s_\mathcal{D}$ be the minimum support used when mining $\mathcal{D}$, and F_D be the set of frequent itemsets obtained. Let Ω be the information kept from the current mining that will be used to enhance the next mining operation. In our case, Ω consists of $F_\mathcal{D}$ (i.e., all frequent itemsets, along with their support counts, in $\mathcal{D}$). An itemset C is frequent in Δ if $\sigma(C) \geq s_\Delta$. Note that an itemset C not frequent in $\mathcal{D}$, may become a frequent itemset in Δ. In this case, C is called an *emerged* itemset. If a frequent itemset in $\mathcal{D}$ remains frequent in Δ it is called a *retained* itemset.

The dataset Δ can be divided into n partitions, $\delta_1, \delta_2, ..., \delta_n$. Each partition δ_i is assigned to a site S_i. We say that Δ is horizontally distributed if its transactions are distributed among the sites. In this case, let $C.sup$ and $C.sup_i$ be the respective support counts of C in Δ and δ_i. We will call $C.sup$ the *global support count* of C, and $C.sup_i$ the *local support count* of C in δ_i. For a given minimum support 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_Δ, and the set of maximal local frequent itemsets at δ_i is denoted as MFI_{δ_i}. The task of mining frequent itemsets in distributed and dynamic datasets is to find F_Δ, with respect to a minimum support s_Δ and, more importantly, using Ω and minimizing access to $\mathcal{D}$ (the original dataset) to enhance the algorithm's performance.

1.2 Related Work

Incremental Mining Some recent effort has been devoted to the problem of incrementally mining frequent itemsets [9,14,15,4,6]. Some of these algorithms cope with the problem of determining when to update the current model, while others update the model after an arbitrary number of updates [15]. To decide when to update, Lee and Cheung [9] propose the DELI algorithm, which uses statistical sampling methods to determine when the current model is outdated. A similar approach proposed by Ganti *et al* [6] monitors changes in the data stream. An efficient incremental algorithm, called ULI, was proposed by Thomas [14] *et al*. ULI strives to reduce the I/O requirements for updating the set of frequent itemsets by maintaining the previous frequent itemsets and the *negative border* [9] along with their support counts. The whole dataset is scanned just once, but the incremental dataset must be scanned as many times as the size of the longest frequent itemset. This work presents extensions to the ZIGZAG algorithm for incremental mining, presented in [15].

Parallel and Distributed Mining Often, the size of a new block of data (i.e., d^+) or the rate at which it is inserted is so large that existing incremental algorithms are ineffective. In these cases, parallel or distributed algorithms are necessary. In [12] a parallel incremental method for performing 2-dimensional discretization on a dynamic dataset was presented. [10] gives an overview of a wide variety of distributed data mining algorithms for collective data mining, clustering, and association rule mining. In particular, there has been much research into parallel and distributed algorithms for mining association rules [16,8,11]. In [17] an overview of several of these methods was presented. In [1] three parallel versions of the apriori algorithm were introduced, namely, COUNT DISTRIBUTION, DATA DISTRIBUTION, and CANDIDATE DISTRIBUTION. They conclude that the COUNT DISTRIBUTION approach performs the best. The FDM (FAST DISTRIBUTED MINING) [3] and DMA (DISTRIBUTED MINING OF ASSOCIATION RULES) [5] algorithms result in fewer candidate itemsets and smaller message sizes compared to the COUNT DISTRIBUTION algorithm. Schuster and Wolff propose the DDM (DISTRIBUTED DECISION MINER) algorithm in [13]. They show that DDM is more scalable than COUNT DISTRIBUTION and FDM with respect to the number of sites participating and the minimum support.

2 Distributed, Parallel and Incremental Algorithm

In this section we describe the algorithm developed to solve the problems defined in the previous section. We start by presenting our incremental algorithm, ZIGZAG. Next, we present a parallel approach for mining the maximal frequent itemsets. Finally, we describe our distributed, parallel and incremental algorithm. We also prove the correctness of the algorithm and present an upper bound for the amount of communication necessary in the distributed and incremental mining operation.

Incremental Algorithm Almost all algorithms for mining frequent itemsets use the same procedure $-$ first a set of candidates is generated, the infrequent ones are pruned, and only the frequent ones are used to generate the next set of candidates. Clearly, an important issue in this task is to reduce the number of candidates generated. An interesting approach to reduce the number of candidates is to first find MFI$_\Delta$. Once MFI$_\Delta$ is found, it is straightforward to obtain all frequent itemsets (and their support counts) in a single dataset scan, without generating infrequent (and unnecessary) candidates. This approach works because the downward closure property (all subsets of a frequent itemset must be frequent). The number of candidates generated by this approach is generally much smaller than the number of candidates generated to directly find all frequent itemsets. The maximal frequent itemsets has been successfully used in several data mining tasks, including incremental mining of evolving datasets [15].

In [15] an efficient incremental algorithm for mining evolving datasets, ZIGZAG, was proposed. The main idea is to incrementally compute MFI$_\Delta$ using previous knowledge Ω. This avoids the generation and testing of many unnecessary candidates. Having MFI$_\Delta$ is sufficient to know which itemsets are frequent; their exact support can be obtained by examining d^+ and using Ω, or, where this is not possible, by examining Δ.

ZIGZAG employs a backtracking search to find MFI$_\Delta$. Backtracking algorithms are useful for many combinatorial problems where the solution can be represented as a set $I = \{i_0, i_1, ...\}$, where each i_j is chosen from a finite *possible set*, P_j. Initially I is empty; it is extended one item at a time, as the search space is traversed. The length of I is the same as the depth of the corresponding node in the search tree. Given a k-candidate itemset, $I_k = \{i_0, i_1, ..., i_{k-1}\}$, the possible values for the next item i_k comes from a subset $R_k \subseteq P_k$ called the *combine set*. If $y \in P_k - R_k$, then nodes in the subtree with root node $I_k = \{i_0, i_1, ..., i_{k-1}, y\}$ will not be considered by the backtracking algorithm. Each iteration of the algorithm tries to extend I_k with every item x in the combine set R_k. An extension is valid if the resulting itemset I_{k+1} is frequent and is not a subset of any already known maximal frequent itemset. The next step is to extract the new possible set of extensions, P_{k+1}, which consists only of items in R_k that follow x. The new combine set, R_{k+1}, consists of those items in the possible set that produce a frequent itemset when used to extend I_{k+1}. Any item not in the combine set refers to a pruned subtree. The backtracking search performs a depth-first traversal of the search space, as depicted in Figure 1. In this example the minimum support is 30%. The framed itemsets are the maximal frequent ones, while the cut itemsets are the infrequent ones.

The support computation employed by ZIGZAG is based on the associativity of itemsets, which is defined as follows. Let C be a k-itemset of items $C_1 \ldots C_k$, where $C_i \in I$. Let $\mathcal{L}(C)$ be its tidset and $\mid \mathcal{L}(C) \mid$ is the length of $\mathcal{L}(C)$ and thus the sup-

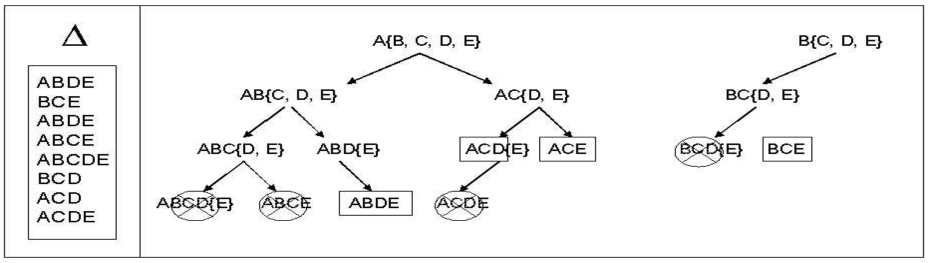

Figure 1. Backtrack Trees for Items A and B on Δ

port count of C. According to [7], any itemset can be obtained by joining its atoms (individual items) and its support count can be obtained by intersecting the tidsets of its subsets. In the first step, ZIGZAG creates a tidset for each item in d^+ and $\mathcal{D}$. The main goal of incrementally computing the support is to maximize the number of itemsets that have their support computed based just on d^+ (i.e., retained itemsets), since their support counts in $\mathcal{D}$ are already stored in Ω. To perform a fast support computation, we first verify if the extension $I_{l+1} \cup \{y\}$ is a retained itemset. If so, its support can be computed by just using d^+ and Ω, thereby enhancing the support computation process.

Parallel Search for Maximal Frequent Itemsets We now consider the problem of parallelizing the search for maximal frequent itemsets in the shared memory paradigm. An efficient parallel search in this paradigm has two main issues: (1) minimizing synchronization, and (2) improving data locality (i.e., maximizing access to local cache). The main idea of our parallel approach is to assign distinct backtrack trees to distinct processors. Note from Figure 1 that the two issues mentioned above can be addressed by this approach. First, there is no dependence among the processors, because each backtrack tree corresponds to a disjoint set of candidates. Since each processor can proceed independently there is no synchronization while searching for maximal frequent itemsets. Second, this approach is very efficient in achieving good data locality, since the support computation of an itemset is based on the intersection of the tidsets of the last two generated subsets. To achieve a suitable level of load-balancing, the backtrack trees are assigned to the processors in a *bag of tasks* approach. That is, we have a *bag* of trees to be processed. Each processor takes one tree, and as soon as it finishes the search on this tree, it takes another task from the *bag*. When the *bag* is empty (i.e., all backtrack trees were processed), all maximal frequent itemsets have been found.

Distributed, Parallel and Incremental Algorithm We now consider the problem of parallelizing the ZIGZAG algorithm in the distributed memory paradigm. We first present Lemma 1, which is the basic theoretical foundation of our distributed approach.

Lemma 1 – *A global frequent itemset must be local frequent in at least one partition.*
Proof. – Let C be an itemset. If $C.sup_i < s_\Delta \times \mid \delta_i \mid$ for all $i = 1, ..., n$, then $C.sup < s_\Delta \times \mid \Delta \mid$ (since $C.sup = \sum_{i=1}^{n} C.sup_i$ and $\mid \Delta \mid = \sum_{i=1}^{n} \mid \delta_i \mid$), and

C cannot be globally frequent. Therefore, if C is a global frequent itemset, it must be local frequent in some partition δ_i. □

In the first step each site S_i independently performs a parallel and incremental search for MFI$_{\delta_i}$, using ZIGZAG on its dataset δ_i. In the second step 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 third 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 support counts 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 (and the support associated with each of these itemsets).

Lemma 2 $-$ $\bigcup_{i=1}^{n}$ *MFI$_{\delta_i}$ determines all global frequent itemsets. Proof.* $-$ We know from Lemma 1 that if C is a global frequent itemset, so it must be local frequent in at least one partition. If C is local frequent in some partition δ_l, so it must be determined by MFI$_{\delta_l}$, and consequently by $\bigcup_{i=1}^{n}$ MFI$_{\delta_i}$. □

By Lemma 2 all global frequent itemsets were found, but not all itemsets generated in the third step are globally frequent (some of them are just locally frequent). The fourth and final step makes a reduction operation on the local support counts of each itemset, to verify which of them are globally frequent in Δ. The process starts with site S_1, which sends the support counts of its itemsets (generated in the third step) to site S_2. Site S_2 sums the support count of each itemset (generated in the third step) with the value of the same itemset obtained from site S_1, and sends the result to site S_3. This procedure continues until site S_n has the global support counts 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 the set of all global frequent itemsets, (i.e., F_{Δ}).

An Upper Bound for the Amount of Communication We also present an upper bound for the amount of communication performed during the distributed mining operation. The upper bound calculation is based just on the local MFIs and on the size of the upper bound for MFI$_{\Delta}$. We divide the upper bound construction into two steps. The first step is related to the local MFI exchange operation. Since each one of the n sites must send its MFI to the other sites, the first term is given by: $\sum_{i=1}^{n}\sum_{j=1}^{|MFI_{\delta_i}|} \mid C_{i,j} \mid$, where $\mid C_{i,j} \mid$ is the size of the j^{th} itemset of the local MFI of site S_i.

The second step is related to the local support count reduction operation. In this operation $n-1$ sites have to pass their local support counts. The amount of communication for this operation is given by: $(n-1) \times \sum_{i=1}^{|UB|}(2^{|C_j|}-1)$, where UB is $\bigcup_{i=1}^{n}$ MFI$_{\delta_i}$, and the term $\sum_{i=1}^{|UB|}(2^{|C_j|}-1)$ represents the local support counts of all subsets of all itemsets in UB. In our data structure a k-itemset is represented by a set of k integers (of 4 bytes). So, in the worst case (when each itemset is subset of only one itemset in UB), the total amount of communication is given by: $(\sum_{i=1}^{n}\sum_{j=1}^{|MFI_{\delta_i}|} \mid C_{i,j} \mid + (n-1) \times \sum_{i=1}^{|UB|}(2^{|C_j|}-1)) \times 4$ bytes. This upper bound shows that our approach is extremely efficient in terms of communication overhead, when compared with the amount of communication necessary to transfer all data among the sites.

3 Experimental Evaluation

Our experimental evaluation was carried out on an 8 node dual PENTIUM processor SMP cluster. Each SMP has 1GB of main memory and 120GB of disk space. We assume that each dataset is already distributed among the nodes, and that updates happen in a periodic fashion. We have implemented the parallel program using the MPI message-passing library (MPICH over GM), and POSIX PTHREADS.

We used both real and synthetic datasets for testing the performance of our algorithm. The WPORTAL dataset is generated from the click-stream log of a large Brazilian web portal, and the WCUP dataset is generated from the click-stream log of the 1998 World Cup web site, which is publicly available at ftp://researchsmp2.cc.vt.edu/pub/. We scanned each log and produced a respective transaction file, where each transaction is a session of access to the site by a client. Each item in the transaction is a web request. Not all requests were turned into items; to become an item, the request must have three properties: (1) the request method is GET; (2) the request status is OK; and (3) the file type is HTML. A session starts with a request that satisfies the above properties, and ends when there has been no click from the client for 30 minutes. All requests in a session must come from the same client. We also used a synthetic dataset, which has been used as benchmarks for testing previous mining algorithms. This dataset mimics the transactions in a retailing environment [2]. WPORTAL has 3,183 distinct items comprised into 7,786,137 transactions (428MB), while WCUP has 5,271 distinct items comprised into 7,618,927 transactions (645MB). T5I2D8000K has 2,000 distinct items comprised into 8,000,000 transactions (1,897MB).

Performance Comparison The first experiment we conducted was to empirically verify the advantages of incremental and parallel mining. We compared the execution time of the distributed algorithms (non-incremental, incremental, parallel non-incremental, and parallel incremental search). We varied the number of nodes (1 to 8), the number of processors per node (1 and 2), and the increment size (10% and 20% of the original dataset). In the incremental case, we first mine a determined number of transactions and then we incrementally mine the remaining transactions. For example, for increment sizes of 20%, we first mine 80% of the dataset and then we incrementally mine the remaining 20%. Each dataset was divided into 1, 2, 4, and 8 partitions, according to the number of nodes employed. Figure 2 shows the execution times obtained for different datasets, and parallel and incremental configurations. As we can see, better execution times are obtained when we combine both parallel and incremental approaches. Further, when the parallel configuration is the same, the execution time is better for smaller increment sizes (since the dataset is smaller), but in some cases the parallel performance is greater than the incremental performance, and better results can be obtained by applying the parallel algorithm with larger increment sizes. This is exactly what happens in the experiments with the WCUP and WPORTAL datasets. The algorithm using the parallel search, applied to an increment size of 20% is more efficient than the algorithm with sequential search, applied to an increment size of 10%, for any number of nodes.

The improvements obtained on the real datasets are not so impressive as the improvement obtained on the synthetic one. The reason is that the real dataset has a skewed data distribution, and therefore the partitions of the real datasets have a very different set of frequent itemsets (and therefore very different local MFIs). On the other hand, the skewness of the synthetic data is very low, therefore each partition of the synthetic dataset is likely to have a similar set of frequent itemsets. From the experiments in the synthetic dataset we observed that $\bigcup_{i=1}^{n} \mathrm{MFI}_{\delta_i}$ was very similar to each local MFI. This means that the set of local frequent itemsets is very similar to the set of global frequent itemsets, and therefore few infrequent candidates are generated by each node.

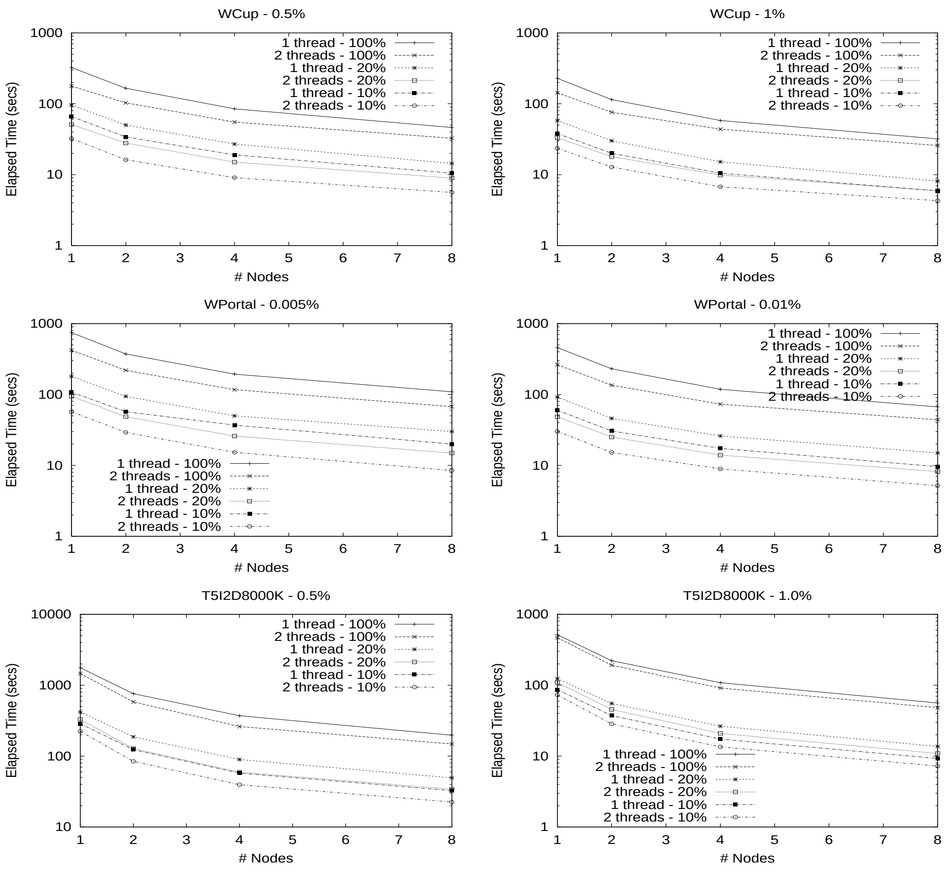

Figure 2. Total Execution Times on different Datasets.

Parallel Performance We also investigated the performance of our algorithm in experiments for evaluating the speedup of different parallel configurations. We used a fixed size dataset with increasing number of nodes. The datasets were divided into 1, 2, 4, and 8 partitions, according to the number of nodes employed. With this configuration we performed speedup experiments on 1, 2, 4, and 8 nodes. In order to evaluate

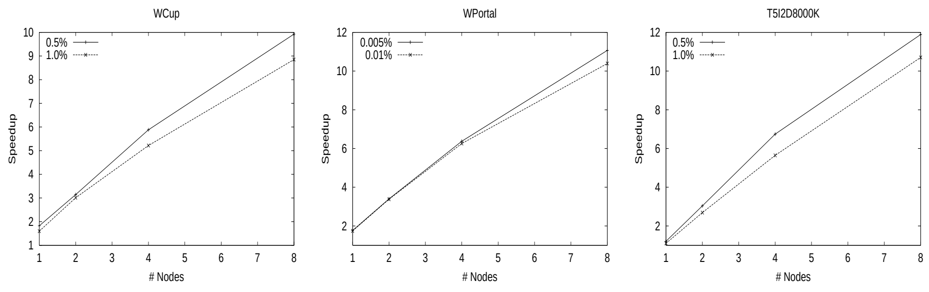

Figure 3. Speedup of the Parallel Algorithm

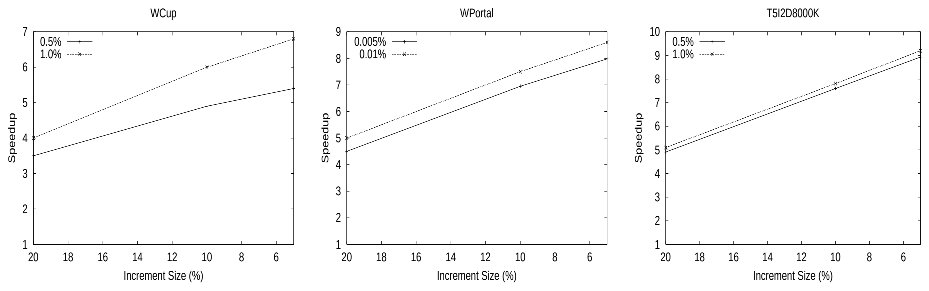

Figure 4. Speedup of the Incremental Algorithm

only the parallel performance, we used the parallel (two processors) non-incremental algorithm and varied the number of nodes. The speedup is in relation to the sequential non-incremental algorithm. Figure 3 shows the speedup numbers of our parallel algorithm. The “super-linear” speedups are due to the parallel MFI search (remember that our environment has 8 dual nodes). Also note that the speedup is inversely proportional to the minimum support. This is because for smaller minimum supports the MFI search becomes more complex, and consequently the parallel task becomes more relevant.

Incremental Performance We also investigated the performance of our algorithm in experiments for evaluating the speedup of different incremental confi gurations. In this experiment, we fi rst mined a fi xed size dataset, and then we performed the incremental mining for different increment sizes (5% to 20%). In order to evaluate only the incremental performance, we used the incremental algorithm with sequential MFI search. We also varied the number of nodes, but the speedup was very similar for different number of nodes, so we show only the results regarding one node. Figure 4 shows the speedup numbers of our incremental algorithm. Note that the speedup is in relation to re-mining the entire dataset. As is expected, the speed is inversely proportional to the size of the increment. This is because the size of the new data coming in is smaller. Also note that better speedups are achieved by greater minimum supports. We observed that, for the datasets used in this experiment, the proportion of retained itemsets (itemsets that are computed by examining only d^+ and Ω) is larger for greater minimum supports.

4 Conclusions

In this paper we considered the problem of mining frequent itemsets on dynamic datasets. We presented an effi cient distributed and parallel incremental algorithm to deal with this problem. Experimental results confi rm that our algorithm results in execution time improvement of more than one order of magnitude when compared against a naive approach. The effi ciency of our algorithm stems from the fact that it makes use of the MFI, reducing both the number of candidates processed and the amount of communication necessary. The MFI is updated by an effi cient parallel and asynchronous backtracking search.

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. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, SanTiago, Chile, June 1994.
3. D. Cheung, J. Han, V. Ng, A. Fu, , and Y. Fu. A fast distributed algorithm for mining association rules. In *4th Int'l. Conf. Parallel and Distributed Info. Systems*, 1996.
4. D. Cheung, S. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. of the 5th Int'l. Conf. on Database Systems for Advanced Applications*, pages 1–4, April 1997.
5. D. Cheung, V. Ng, A. Fu, , and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 911–922, 1996.
6. V. Ganti, J. Gehrke, and R. Ramakrishnan. Demon: Mining and monitoring evolving data. In *Proc. of the 16th Int'l Conf. on Data Engineering*, pages 439–448, San Diego, USA, 2000.
7. K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. of the 1st IEEE Int'l Conf. on Data Mining*, San Jose, USA, November 2001.
8. E.-H. Han, G. Karypis, , and V. Kumar. Scalable parallel data mining for association rules. In *ACM SIGMOD Conf. Management of Data*, 1997.
9. S. Lee and D. Cheung. Maintenance of discovered association rules: When to update? In *Research Issues on Data Mining and Knowledge Discovery*, 1997.
10. Byung-Hoon Park and Hillol Kargupta. Distributed data mining: Algorithms, systems, and applications. In Nong Ye, editor, *Data Mining Handbook*, 2002.
11. J.S. Park, M. Chen, , and P. S. Yu. CACTUS - clustering categorical data using summaries. In *ACM Int'l. Conf. on Information and Knowledge Management*, 1995.
12. S. Parthasarathy and A. Ramakrishnan. Parallel incremental 2d discretization. In *Proc. IEEE Int'l Conf. on Parallel and Distributed Processing*, 2002.
13. A. Schuster and R. Wolff. Communication efficient distributed mining of association rules. In *ACM SIGMOD Int'l. Conf. on Management of Data*, 2001.
14. S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules. In *Proc. of the 3rd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, August 1997.
15. A. Veloso, W. Meira Jr., M. Bunte, S. Parthasarathy, and M. Zaki. Mining frequent itemsets in evolving databases. In *Proc. of the 2nd SIAM Int'l Conf. on Data Mining*, USA, 2002.
16. M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 4(1):343–373, December 1997.
17. M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, December 1999.

View publication stats