Association Rule Mining in Peer-to-Peer Systems*

Ran Wolff and Assaf Schuster
Technion – Israel Institute of Technology
Email: {ranw,assaf}@cs.technion.ac.il

Abstract

We extend the problem of association rule mining – a key data mining problem – to systems in which the database is partitioned among a very large number of computers that are dispersed over a wide area. Such computing systems include GRID computing platforms, federated database systems, and peer-to-peer computing environments. The scale of these systems poses several difficulties, such as the impracticality of global communications and global synchronization, dynamic topology changes of the network, on-the-fly data updates, the need to share resources with other applications, and the frequent failure and recovery of resources.

We present an algorithm by which every node in the system can reach the exact solution, as if it were given the combined database. The algorithm is entirely asynchronous, imposes very little communication overhead, transparently tolerates network topology changes and node failures, and quickly adjusts to changes in the data as they occur. Simulation of up to 10,000 nodes show that the algorithm is local: all rules, except for those whose confidence is about equal to the confidence threshold, are discovered using information gathered from a very small vicinity, whose size is independent of the size of the system.

1 Introduction

The problem of association rule mining (ARM) in large transactional databases was first introduced in 1993 [1]. The input to the ARM problem is a database in which objects are grouped by context. An example would be a list of items grouped by the transaction in which they were bought. The objective of ARM is to find sets of objects which tend to associate with one another. Given two distinct sets of objects, X and Y, we say Y is associated with X if the appearance of X in a certain context usually implies that Y will appear in that context as well. The output of an ARM algorithm is a list of all the association rules that appear frequently in the database and for which the association is confident.

ARM has been the focus of great interest among data mining researchers and practitioners. It is today widely accepted to be one of the key problems in the data mining field. Over the years many variations were described for ARM, and a wide range of applications were developed. The overwhelming majority of these deal with sequential ARM algorithms. Distributed association rule mining (D-ARM) was defined in [2], not long after the definition of ARM, and was also the subject of much research (see, for example, [2, 5, 15, 7, 8]).

In recent years, database systems have undergone major changes. Databases are now detached from the computing servers and have become distributed in most cases. The natural extension of these two changes is the development of federated databases – systems which connect many different databases and present a single database image. The trend toward ever more distributed databases goes hand in hand with an ongoing trend in large organizations toward ever greater integration of data. For example, health maintenance organizations (HMOs) envision their medical records, which are stored in thousands of clinics, as one database. This integrated view of the data is imperative for essential data analysis applications ranging from epidemic control, ailment and treatment pattern discovery, and the detection of medical fraud or misconduct. Similar examples of this imperative are common in other fields, including credit card companies, large retail networks, and more.

An especially interesting example for large scale distributed databases are peer-to-peer systems. These systems include GRID computing environments such as Condor [10] (20,000 computers), specific area computing systems such as SETI@home [12] (1,800,000 computers) or UnitedDevices [14] (2,200,000 computers), general purpose peer-to-peer platforms such as Entropia [6] (60,000 peers), and file sharing networks such as Kazza (1.8 million peers). Like any other system, large scale distributed systems maintain and produce operational data. However,

*This work was supported in part by Microsoft Academic Foundation.

in contrast to other systems, that data is distributed so widely that it will usually not be feasible to collect it for central processing. It must be processed in place by distributed algorithms suitable to this kind of computing environment.

Consider, for example, mining user preferences over the Kazza file sharing network. The files shared through Kazza are usually rich media files such as songs and videos. participants in the network reveal the files they store on their computers to the system and gain access to files shared by their peers in return. Obviously, this database may contain interesting knowledge which is hard to come by using other means. It may be discovered, for instance, that people who download The Matrix also look for songs by Madonna. Such knowledge can then be exploited in a variety of ways, much like the well known data mining example stating that "customers who purchase diapers also buy beer".

The large-scale distributed association rule mining (LSD-ARM) problem is very different from the D-ARM problem, because a database that is composed of thousands of partitions is very different from a small scale distributed database. The scale of these systems introduces a plethora of new problems which have not yet been addressed by any ARM algorithm. The first such problem is that in a system that large there can be no global synchronization. This has two important consequences for any algorithm proposed for the problem: The first is that the nodes must act independently of one another; hence their progress is speculative, and intermediate results may be overturned as new data arrives. The second is that there is no point in time in which the algorithm is known to have finished; thus, nodes have no way of knowing that the information they possess is final and accurate. At each point in time, new information can arrive from a far-away branch of the system and overturn the node's picture of the correct result. The best that can be done in these circumstances is for each node to maintain an assumption of the correct result and update it whenever new data arrives. Algorithms that behave this way are called *anytime algorithms*.

Another problem is that global communication is costly in large scale distributed systems. This means that for all practical purposes the nodes should compute the result through local negotiation. Each node can only be familiar with a small set of other nodes – its immediate neighbors. It is by exchanging information with their immediate neighbors concerning their local databases that nodes investigate the combined, global database.

A further complication comes from the dynamic nature of large scale systems. If the mean time between failures of a single node is 20,000 hours[1], a system consisting of 100,000 nodes could easily fail five times per hour. Moreover, many such systems are purposely designed to support the dynamic departure of nodes. This is because a system that is based on utilizing free resources on non-dedicated machines should be able to withstand scheduled shutdowns for maintenance, accidental turnoffs, or an abrupt decrease in availability when the user comes back from lunch. The problem is that whenever a node departs, the database on that node may disappear with it, changing the global database and the result of the computation. A similar problem occurs when nodes join the system in mid-computation.

Obviously none of the distributed ARM algorithms developed for small-scale distributed systems can manage a system with the aforementioned features. These algorithms focus on achieving parallelization induced speedups. They use basic operators, such as broadcast, global synchronization, and a centralized coordinator, none of which can be managed in large-scale distributed systems. To the best of our knowledge, no D-ARM algorithm presented so far acknowledges the possibility of failure. Some relevant work was done in the context of incremental ARM, e.g., [13], and similar algorithms. In these works the set of rules is adjusted following changes in the database. However, we know of no parallelizations for those algorithms even for small-scale distributed systems.

In this paper we describe an algorithm which solves LSD-ARM. Our first contribution is the inference that the distributed association rule mining problem is reducible to the well-studied problem of distributed majority votes. Building on this inference, we develop an algorithm which combines sequential association rule mining, executed locally at each node, with a majority voting protocol to discover, at each node, all of the association rules that exist in the combined database. During the execution of the algorithm, which in a dynamic system may never actually terminate, each node maintains an ad hoc solution. If the system remains static, then the ad hoc solution of most nodes will quickly converge toward an exact solution. This is the same solution that would be reached by a sequential ARM algorithm had all the databases been collected and processed. If the static period is long enough, then all nodes will reach this solution. However, in a dynamic system, where nodes dynamically join or depart and the data changes over time, the changes are quickly and locally adjusted to, and the solution continues to converge. It is worth mentioning that no previous ARM algorithm was proposed which mines rules (not itemsets) on the fly. This contribution may affect other kinds of ARM algorithms, especially those intended for data streams [9].

The majority voting protocol, which is at the crux of our algorithm, is in itself a significant contribution. It requires no synchronization between the computing nodes.

[1] This figure is accepted for hardware; for software the estimate is usually a lot lower.

Each node communicates only with its immediate neighbors. Moreover, the protocol is local: in the overwhelming majority of cases, each node computes the majority – i.e., identifies the correct rules – based upon information arriving from a very small surrounding environment. Locality implies that the algorithm is scalable to very large networks. Another outcome of the algorithm's locality is that the communication load it produces is small and roughly uniform, thus making it suitable for non-dedicated environments.

2 Problem Definition

The association rule mining (ARM) problem is traditionally defined as follows: Let $I = \{i_1, i_2, ..., i_m\}$ be the items in a certain domain. An itemset is some subset $X \subseteq I$. A transaction t is also a subset of I associated with a unique transaction identifier TID. A database DB is a list that contains $|DB|$ transactions. Given an itemset X and a database DB, $Support(X, DB)$ is the number of transactions in DB which contain all the items of X, and $Freq(X, DB) = Support(X, DB) / |DB|$. For some frequency threshold $0 \leq MinFreq \leq 1$, we say that an itemset X is frequent in a database DB if $Freq(X, DB) \geq MinFreq$ and infrequent otherwise. For two distinct frequent itemsets X and Y, and a confidence threshold $0 \leq MinConf \leq 1$, we say the rule $X \Rightarrow Y$ is confident in DB if $Freq(X \cup Y, DB) \geq MinConf \cdot Freq(X, DB)$. We will call confident rules between frequent itemsets correct and the remaining rules false. The solution for the ARM problem is $R[DB]$, the list of all the correct rules in the given database.

When the database is dynamically updated, that is, transactions are added to it or deleted from it over time, we denote DB_t the database at time t. Now consider that the database is also partitioned among an unknown number of share nothing machines (nodes); we denote the partition of node u at time t DB_t^u. Given an infrastructure through which those machines may communicate data, we denote $[u]_t$ the group of machines reachable from u at time t. We will assume symmetry, i.e., $v \in [u]_t \Leftrightarrow u \in [v]_t$. Nevertheless, $[u]_t$ may or may not include all of the machines. The solution to the large-scale distributed association rule mining (LSD-ARM) problem for node u at time t is the set of rules which are correct in the combined databases of the machines in $[u]_t$; we denote this solution $R[u]_t$.

Since both $[u]_t$ and DB_t^v of each $v \in [u]_t$ are free to vary with time – so does $R[u]_t$. It is thus imperative that u calculate not only the eventual solution but also approximated, ad hoc, solutions. We term $\tilde{R}[u]_t$ the ad hoc solution of node u at time t. It is common practice to measure the performance of an anytime algorithm according to its recall: $\frac{|R[u]_t \cap \tilde{R}[u]_t|}{|R[u]_t|}$, and its precision: $\frac{|R[u]_t \cap \tilde{R}[u]_t|}{|\tilde{R}[u]_t|}$.

We require that if both $[u]_t$ and DB_t^v of each $v \in [u]_t$ remain static long enough, then the approximated solution $\tilde{R}[u]_t$ will converge to $R[u]_t$. In other words, both the precision and the recall of u converge to a hundred percent.

Throughout this work we make some simplifying assumptions. We assume that node connectivity is a forest – for each u and v there could be either one route from u to v or none. Trees in the forest may split, join, grow, and shrink, as a result of node crash and recovery, or departure and join. We assume the failure model of computers is fail-stop, and that a node is informed of changes in the status of adjacent nodes.

3 An ARM Algorithm for Large-Scale Distributed Systems

As previously described, our algorithm is comprised of two rather independent components: Each node executes a sequential ARM algorithm which traverses the local database and maintains the current result. Additionally, each node participates in a distributed majority voting protocol which makes certain that all nodes that are reachable from one another converge toward the correct result according to their combined databases. We will begin by describing the protocol and then proceed to show how the full algorithm is derived from it.

3.1 LSD-Majority Protocol

It has been shown in [11] that a distributed ARM algorithm can be viewed as a decision problem in which the participating nodes must decide whether or not each itemset is frequent. However, the algorithm described in that work extensively uses broadcast and global synchronization; hence it is only suitable for small-scale distributed systems. We present here an entirely different majority voting protocol – LSD-Majority – which works well for large-scale distributed systems. In the interest of clarity we describe the protocol assuming the data at each node is a single bit. We will later show how the protocol can easily be generalized for frequency counts.

As in LSD-ARM, the purpose of LSD-Majority is to ensure that each node converges toward the correct majority. Since the majority problem is binary, we measure the recall as the proportion of nodes u whose ad hoc solution is one when the majority in $[u]_t$ is of set bits, or zero when the majority in $[u]_t$ is of unset bits. The protocol dictates how nodes react when the data changes, a message is received, or a neighboring node is reported to have detached or joined.

The nodes communicate by sending messages that contain two integers: $count$, which stands for the number of bits this message reports, and sum which is the number of those bits which are equal to one. Each node u will record, for every neighbor v, the last message it sent to v – $\{sum^{uv}, count^{uv}\}$ – and the last message it received from v – $\{sum^{vu}, count^{vu}\}$. Node u calculates the following two functions of these messages and its own local bit:

$$\Delta^u = s^u + \sum_{vu \in E^u} sum^{vu} - \lambda \left(c^u + \sum_{vu \in E^u} count^{vu} \right)$$

$$\Delta^{uv} = sum^{uv} + sum^{vu} - \lambda \left(count^{uv} + count^{vu} \right)$$

Here E^u is the set of edges colliding with u, s^u is the value of the local bit, and c^u is, for now, one. Δ^u measures the number of access set bits u has been informed of. Δ^{uv} measures the number of access set bits u and v have last reported to one another. Each time s^u changes, a message is received, or a node connects to v or disconnects from v, Δ^u is recalculated; Δ^{uv} is recalculated each time a message is sent to or received from v.

Algorithm 1 LSD-Majority

Input for node u: The set of edges that collide with it E^u, A bit s^u and the majority ratio λ.
Output: The algorithm never terminates. Nevertheless, at each point in time if $\Delta^u \geq 0$ then the output is 1, otherwise it is 0.
Definitions: $\Delta^u = s^u + \sum_{vu \in E^u} sum^{vu} - \lambda \left(c^u + \sum_{vu \in E^u} count^{vu} \right)$, $\Delta^{uv} = sum^{uv} + sum^{vu} - \lambda \left(count^{uv} + count^{vu} \right)$
Initialization: For each $vu \in E^u$ set sum^{vu}, $count^{vu}$, sum^{uv}, $count^{uv}$ to 0. Set $c^u = 1$.
On edge vu recovery : Add vu to E^u. Set sum^{vu}, $count^{vu}$, sum^{uv}, $count^{uv}$ to 0.
On failure of edge $vu \in E^u$: Remove vu from E^u.
On message $\{sum, count\}$ received over edge vu: Set $sum^{vu} = sum$, $count^{vu} = count$
On change in s^u, edge failure or recovery, or the receiving of a message:
For each $vu \in E^u$
If $count^{uv} + count^{vu} = 0$ and $\Delta^u \geq 0$
or $count^{uv} + count^{vu} > 0$ and either $\Delta^{uv} < 0$ and $\Delta^u > \Delta^{uv}$ or $\Delta^{uv} \geq 0$ and $\Delta^u < \Delta^{uv}$
Set $sum^{uv} = s^u + \sum_{wu \neq vu \in E^u} sum^{wu}$ and $count^{uv} = c^u + \sum_{wu \neq vu \in E^u} count^{wu}$
Send $\{sum^{uv}, count^{uv}\}$ over vu to v

Each node performs the protocol independently with each of its immediate neighbors. Node u coordinates its majority decision with node v by maintaining the same Δ^{uv} value (note that $\Delta^{uv} = \Delta^{vu}$) and making certain that Δ^{uv} will not mislead v into believing that the global majority is larger than it actually is. As long as $\Delta^u \geq \Delta^{uv} \geq 0$ and $\Delta^v \geq \Delta^{vu} \geq 0$, there is no need for u and v to exchange data. They both calculate the majority of the bits to be set; thus, the majority in their combined data must be of set bits. If, on the other hand, $\Delta^{uv} > \Delta^u$, then v might mistakenly calculate $\Delta^v \geq 0$ because it has not received the updated data from u. Thus, in this case the protocol dictates that u send v a message, $\left\{ s^u + \sum_{wu \neq vu \in E^u} sum^{wu}, c^u + \sum_{wu \neq vu \in E^u} count^{wu} \right\}$. Note that after this message is sent, $\Delta^{uv} = \Delta^u$.

The opposite case is almost the same. Again, if $0 > \Delta^{uv} \geq \Delta^u$ and $0 > \Delta^{vu} \geq \Delta^v$, then no messages are exchanged. However, when $\Delta^u > \Delta^{uv}$, the protocol dictates that u send v a message calculated the same way. The only difference is that when no messages were sent or received, v knows, by default, that $\Delta^u < 0$ and u knows that $\Delta^v < 0$. Thus, unless $\Delta^u \geq 0$, u does not send messages to v because the majority bits in their combined data cannot be set.

The pseudocode of the LSD-Majority protocol is given in Algorithm 1. It is easy to see that when the protocol dictates that no node needs to send any message, either $\Delta^v \geq 0$ for all nodes $v \in [u]_t$, or $\Delta^v < 0$ for all of them. If there is disagreement in $[u]_t$, then there must be disagreement between two immediate neighbors, in which case at least one node v must send data, which will cause $count^{uv} + count^{vu}$ to increase. This number is bounded by the size of $[u]_t$; hence, the protocol always reaches consensus in a static state. It is less trivial to show that the conclusion they arrive at is the correct one. This proof is too long to include in this context.

In order to generalize LSD-Majority for frequency counts, c^u need only to be set to the size of the local database and s^u to the local support of an itemset. Then, if we substitute $MinFreq$ for λ, the resulting protocol will decide whether an itemset is frequent or infrequent in $[u]_t$. Deciding whether a rule $X \Rightarrow Y$ is confident is also straightforward using this protocol: c^u should now count in the local database the number of transactions that include X, s^u should count the number of these transactions that include both X and Y, and λ should be replaced with $MinConf$.

Deciding whether a rule is correct or false requires that each node run two instances of the protocol: one to decide whether the rule is frequent and the other to decide whether it is confident. Note, however, that for all rules of the form $A \Rightarrow B \setminus A$, only one instance of the protocol should be performed to decide whether B is frequent.

The strength of the protocol lies in its behavior when

the average of the bits over $[u]_t$ is somewhat different than the majority threshold λ. Defining the significance of the input as $\frac{\sum_{v\in[u]_t} s^v}{\lambda\cdot\sum_{v\in[u]_t} c^v} - 1$, we will show in section 4.1 that even a minor significance, on the scale of ± 0.1, is sufficient for making a correct decision using data from just a small number of nodes. In other words, even a minor significance is sufficient for the algorithm to become local. Another strength of the protocol is that during static periods, most of the nodes will make the correct majority decision very quickly. These two features make LSD-Majority especially well-suited for LSD-ARM, in which the overwhelming majority of the candidates are far from significant.

3.2 Majority-Rule Algorithm

LSD-Majority efficiently decides whether a candidate rule is correct or false. It remains to show how candidates are generated and how they are counted in the local database. The full algorithm must satisfy two requirements: First, each node must take into account not only the local data, but also data brought to it by LSD-Majority, as this data may indicate that additional rules are correct and thus further candidates should be generated. Second, unlike other algorithms, which produce rules after they have finished discovering all itemsets, an algorithm which never really finishes discovering all itemsets must generate rules on the fly. Therefore the candidates it uses must be rules, not itemsets. We now present an algorithm – Majority-Rule – which satisfies both requirements.

The first requirement is rather easy to satisfy. We simply increment the counters of each rule according to the data received. Additionally, we employ a candidate generation approach that is not levelwise: as in the DIC algorithm [4], we periodically consider all the correct rules, regardless of when they were discovered, and attempt to use them for generating new candidates.

The second requirement, mining rules directly rather than mining itemsets first and producing rules when the algorithm terminates, has not, to the best of our knowledge, been addressed in the literature. To satisfy this requirement we generalize the candidate generation procedure of Apriori [3]. Apriori generates candidate itemsets in two ways: Initially, it generates candidate itemsets of size 1: $\{i\}$ for every $i \in I$. Later, candidates of size $k+1$ are generated by finding pairs of frequent itemsets of size k that differ by only the last item – $X \cup \{i_1\}$ and $X \cup \{i_2\}$ – and validating that all of the subsets of $X \cup \{i_1, i_2\}$ are also frequent before making that itemset a candidate. In this way, Apriori generates the minimal candidate set which must be generated by any deterministic algorithm.

In the case of Majority-Rule, the dynamic nature of the

Algorithm 2 Majority-Rule

Input for node u: The set of edges that collide with it E^u. The local database DB^u. $MinFreq$, $MinConf$, M
Initialization: Set $C \leftarrow \{\langle \emptyset \Rightarrow \{i\}\rangle \ for\ all\ i \in I\}$
For each $r \in C$ set $r.sum = r.count = 0$, and $r.\lambda = MinFreq$
For each $r \in C$ and every $vu \in E^u$ set $r.sum^{uv} = r.count^{uv} = r.sum^{vu} = r.count^{vu} = 0$
Upon receiving $\{r.id, sum, count\}$ **from a neighbor** v If $r = \langle X \Rightarrow Y\rangle \notin C$ add it to C. If $c' = \langle \emptyset \Rightarrow X \cup Y\rangle \notin C$ add it too.
Set $r.sum^{vu} = sum$, $r.count^{vu} = count$
On edge vu **recovery:** Add vu to E^u. For all $r \in C$ set $r.sum^{uv} = r.count^{uv} = r.sum^{vu} = r.count^{vu} = 0$
On failure of edge $vu \in E^u$**:** Remove vu from E^u.
Main: Repeat the following for ever
Read the next transaction – T. If it is the last one in DB^u iterate back to the first one.
For every $r = \langle X \Rightarrow Y\rangle \in C$ which was generated after this transaction was last read
If $X \subseteq T$ increase $r.count$
If $X \cup Y \subseteq T$ increase $r.sum$
Once every M transactions
Set $\tilde{R}[u]_l$ = the set of rules $r = \langle X \Rightarrow Y\rangle \in C$ such that $\Delta^u(r) \geq 0$ and for $r' = \langle \emptyset \Rightarrow X \cup Y\rangle$ $\Delta^u(r') \geq 0$
For every $r = \langle X \Rightarrow Y\rangle \in \tilde{R}[u]_l$, such that $X = \emptyset$ and $i \in X$ if $r' = \langle X \setminus \{i\} \Rightarrow \{i\}\rangle \notin C$ insert r' into C with $r'.sum = r'.count = 0$, $r'.\lambda = MinConf$ and $r'.id =$ unique rule id
For each $c_1 = \langle X \Rightarrow Y \cup \{i_1\}\rangle, c_2 = \langle X \Rightarrow Y \cup \{i_2\}\rangle \in \tilde{R}[u]_t$ such that $i_1 < i_2$, if $c_3 = \langle X \Rightarrow Y \cup \{i_1, i_2\}\rangle \notin C$ and $\forall i_3 \in Y : r_3 = \langle X \Rightarrow Y \cup \{i_1, i_2\} \setminus \{i_3\}\rangle \in \tilde{R}[u]_t$, add r_3 to C with $r_3.sum = r_3.count = 0$, $r_3.\lambda = r_1.\lambda$, and $r_3.id =$ unique rule id
For each $r = \langle X \Rightarrow Y\rangle \in C$ and for every $vu \in E^u$
If $r.count^{uv} + r.count^{vu} = 0$ and $\Delta^u(r) \geq 0$
or $r.count^{uv} + r.count^{vu} > 0$ and either $\Delta^{uv}(r) < 0$ and $\Delta^u(r) > \Delta^{uv}(r)$
or $\Delta^{uv} \geq 0$ and $\Delta^u(r) < \Delta^{uv}(r)$
Set $r.sum^{uv} = r.sum + \sum_{wu\neq vu\in E^u} r.sum^{wu}$ and $r.count^{uv} = r.count + \sum_{wu\neq vu\in E^u} r.count^{wu}$
Send $\{r.id, r.sum^{uv}, r.count^{uv}\}$ over vu to v

system means that it is never certain whether an itemset is frequent or a rule is correct. Thus, it is impossible to guarantee that no superfluous candidates are generated. Nevertheless, at any point during execution t, it is worthwhile to use the ad hoc set of rules, $\tilde{R}[u]_t$, to try and limit the number of candidate rules. Our candidate generation criterion is thus a generalization of Apriori's criterion. Each node generates initial candidate rules of the form $\emptyset \Rightarrow \{i\}$ for each $i \in I$. Then, for each rule $\emptyset \Rightarrow X \in \tilde{R}[u]_t$, it generates $X \setminus \{i\} \Rightarrow \{i\}$ candidate rules for all $i \in X$. In addition to these initial candidate rules, the node will look for pairs of rules in $R[u]_t$ which have the same left-hand side, and right-hand sides that differ only in the last item – $X \Rightarrow Y \cup \{i_1\}$ and $X \Rightarrow Y \cup \{i_2\}$. The node will verify that the rules $X \Rightarrow Y \cup \{i_1, i_2\} \setminus \{i_3\}$, for every $i_3 \in Y$, are also correct, and then generate the candidate $X \Rightarrow Y \cup \{i_1, i_2\}$. It can be inductively proved that if $\tilde{R}[u]_t$ contains only correct rules, then no superfluous candidate rules are ever generated using this method.

The rest of Majority-Rule is straightforward. Whenever a candidate is generated, the node will begin to count its support and confidence in the local database. At the same time, the node will also begin two instances of LSD-Majority, one for the candidate's frequency and one for its confidence, and these will determine whether this rule is globally correct. Since each node runs multiple instances of LSD-Majority concurrently, messages must carry, in addition to sum and $count$, the identification of the rule it refers to, $r.id$. We will denote $\Delta^u(r)$ and $\Delta^{uv}(r)$ the result of the previously defined functions when they refer to the counters and λ of candidate r. Finally, $r.\lambda$ is the majority threshold that applies to r. We set $r.\lambda$ to $MinFreq$ for rules with an empty left-hand side and to $MinConf$ for all other rules.

The pseudocode of Majority-Rule is detailed in Algorithm 2.

4 Experimental Results

To evaluate Majority-Rule's performance, we implemented a simulator capable of running thousands of simulated computers. We simulated 1600 such computers, connected in a random tree overlaid on a 40×40 grid. We also implemented a simulator for a stand-alone instance of the LSD-Majority protocol and ran simulations of up to 10,000 nodes on a 100×100 grid. The simulations were run in lock-step, not because the algorithm requires that the computers work in locked-step – the algorithm poses no such limitations – but rather because properties such as convergence and locality are best demonstrated when all processors have the same speed and all messages are delivered in unit time.

We used synthetic databases generated by the standard tool from the IBM-quest data mining group [3]. We generated three synthetic databases – T5.I2, T10.I4 and T20.I6 – where the number after T is the average transaction length and the number after I is the average pattern length. The combined size of each of the three databases is 10,000,000 transactions. Other than the number of transactions the change we made from the defaults was reducing the number of patterns. This was reduced so as to increase the proportion of correct rules from one in ten-thousands to one in a hundred candidates. Because our algorithm performs better for false rules than for correct ones this change does not impair out the validity of the results.

4.1 Locality of LSD-Majority and Majority-Rule

The LSD-Majority protocol, and consequently the Majority-Rule algorithm, are local algorithms in the sense that a correct decision will usually only require that a small subset of the data is gathered. We measure the locality of an algorithm by the average and maximum size of the environment of nodes. The environment is defined in LSD-Majority as the number of input bits received by the node and in Majority-Rule as the percent of the global database reported to the node, until system stabilization. Our experiments with LSD-Majority show that its locality strongly depends on the significance of the input: $\frac{\sum_{v \in [u]_t} s^v}{\lambda \cdot \sum_{v \in [u]_t} c^v} - 1$.

Figure 1(a) describes the results of a simulation of 10,000 nodes in a random tree over a grid, with various percentages of set input bits at the nodes. It shows that when the significance is ± 0.1 (i.e., 45% or 55% of the nodes have set input bits), the protocol already has good locality: the maximal environment is about 1200 nodes and the average size a little over 300. If the percentage of set input bits is closer to the threshold, a large portion of the data would have to be collected in order to find the majority. In the worst possible case, when the number of set input bits is equal to the number of unset input bits plus one, at least one node would have to collect all of the input bits before the solution could be reached. On the other hand, if the percentage of set input bits is further from the threshold, then the average environment size becomes negligible. In many cases different regions of the grid may not exchange any messages at all. In Figure 1(c) these results repeat themselves for Majority-Rule.

Further analysis – Figure 1(c) – show that the size of a node's environment depends on the significance in a small region around the nodes. I.e., if the inputs of nodes are independent of one another then the environment size will be random. This makes our algorithms fair: nodes' performance is not determined by its connectivity or location but rather by the data.

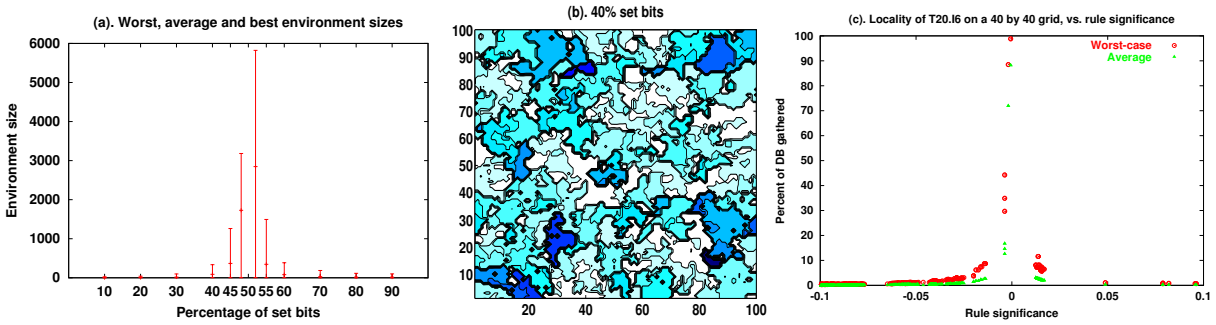

Figure 1. The locality of LSD-Majority (a) and of Majority-Rule (c) depends of the significance. The distribution of environment sizes (b) depends on the local significance and is hence random.

4.2 Convergence and Cost of Majority-Rule

In addition to locality, the other two important characteristics of Majority-Rule are its convergence rate and communication cost. We measure convergence by calculating the recall – the percentage of rules uncovered – and precision – the percentage of correct rules among all rules assumed correct – vis-a-vis the number of transactions scanned. Figure 2 describes the convergence of the recall (a) and of the precision (b). In (c) the convergence of stand-alone LSD-Majority is given, for various percentages of set input bits.

To understand the convergence of Majority-Rule, one must look at how the candidate generation and the majority voting interact. Rules which are very significant are expected to be generated early and agreed upon fast. The same holds for false candidates with extremely low significance. They too are generated early, because they are usually generated due to noise, which subsides rapidly as a greater portion of the local database is scanned; the convergence of LSD-Majority will be as quick for them as for rules with high significance. This leaves us with the group of marginal candidates, those that are very near to the threshold; these marginal candidates are hard to agree upon, and in some cases, if one of their subsets is also marginal, they may only be generated after the algorithm has been working for a long time. We remark that marginal candidates are as difficult for other algorithms as they are for Majority-Rule. For instance, DIC may suffer from the same problem: if all rules were marginal, then the number of database scans would be as large as that of Apriori.

An interesting feature of LSD-Majority convergence is that the number of nodes that assume a majority of set bits always increases in the first few rounds. This would result in a sharp reduction in accuracy in the case of a majority of unset bits, and an overshot, above the otherwise exponential convergence, in the case of a majority of set bits. This occurs because our protocol operates in expanding wavefronts, convincing more and more nodes that there is a certain majority, and then retreating with many nodes being convinced that the majority is the opposite. Since we assume by default a majority of zeros, the first wavefront that expands would always be about a majority of ones. Interestingly enough, the same pattern can be seen in the convergence of Majority-Rule (more clearly for the precision than for the recall).

Figure 3 presents the the communication cost of LSD-Majority vis-a-vis the percentage of set input bits and of Majority-Rule vis-a-vis rule significance. For rules that are very near the threshold, a lot of communication is required, on the scale of the grid diameter. For significant rules the communication load is about ten messages per rule per node. However, for false candidates the communication load drops very fast to nearly no messages at all. It is important to keep in mind that we denote every pair of integers we send a message. In a realistic scenario, a message will contain up to 1500 bytes, or about 180 integer pairs.

5 Conclusions

We have described a new distributed majority vote protocol – LSD-Majority– which we incorporated as part of an algorithm – Majority-Rule – that mines association rules on distributed systems of unlimited size. We have shown that the key quality of our algorithm is its locality – the fact that information need not travel far on the network for the correct solution to be reached. We have also shown that the locality of Majority-Rule translates into fast convergence of the result and low communication demands. Communication is also very efficient, at least for candidate rules which turn out not to be correct. Since the overwhelming majority of the candidates usually turn out this way, the communication load of Majority-Rule depends mainly

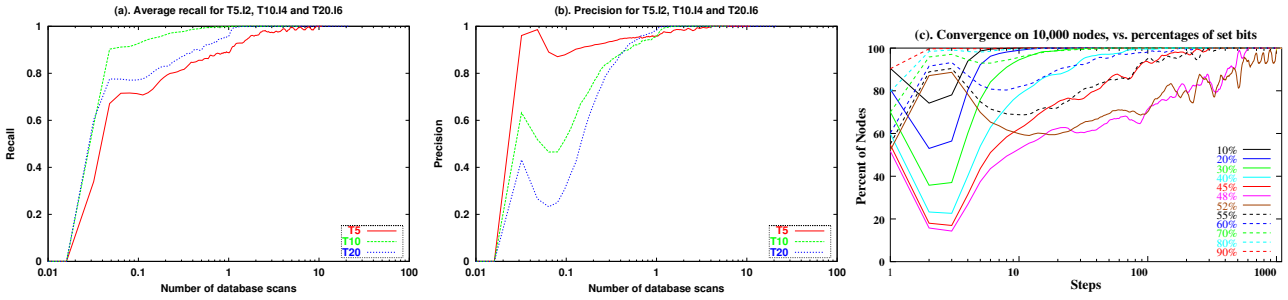

Figure 2. Convergence of the recall and precision of Majority-Rule, and of LSD-Majority.

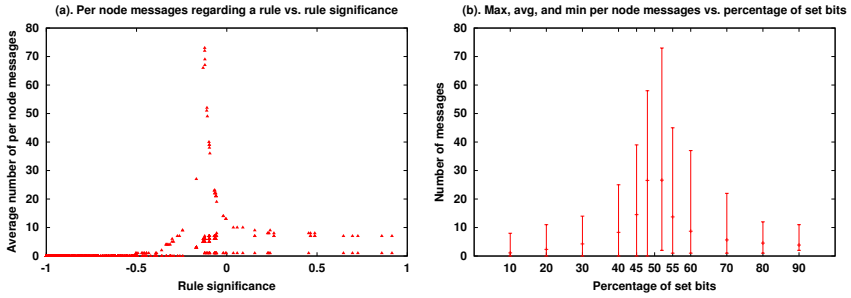

Figure 3. Communication characteristics of Majority-Rule (a), and of LSD-Majority (b). Each message here is a pair of integers.

on the size of the output – the number of correct rules. That number is controllable via user supplied parameters, namely $MinFreq$ and $MinConf$.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Int'l. Conference on Management of Data*, pages 207–216, Washington, D.C., June 1993.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l. Conference on Very Large Databases (VLDB'94)*, pages 487 – 499, Santiago, Chile, September 1994.

[4] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Record*, 6(2):255–264, June 1997.

[5] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proc. of 1996 Int'l. Conf. on Parallel and Distributed Information Systems*, pages 31 – 44, Miami Beach, Florida, December 1996.

[6] Entropia. `http://www.entropia.com`.

[7] E.-H. S. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):352 – 377, 2000.

[8] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proceedings of the 14th Int'l. Conference on Data Engineering (ICDE'98)*, pages 486–493, 1998.

[9] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[10] T. C. Project. `http://www.cs.wisc.edu/condor/`.

[11] A. Schuster and R. Wolff. Communication-efficient distributed mining of association rules. In *Proc. of the 2001 ACM SIGMOD Int'l. Conference on Management of Data*, pages 473 – 484, Santa Barbara, California, May 2001.

[12] Seti@home. `http://setiathome.ssl.berkeley.edu/`.

[13] S. Thomas and S. Chakravarthy. Incremental mining of constrained associations. In *HiPC*, pages 547–558, 2000.

[14] United devices inc. http://www.ud.com/home.htm.

[15] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.