Association Rule Mining in Peer-to-Peer Systems

Ran Wolff and Assaf Schuster
Computer Science Department
Technion – Israel Institute of Technology
E-mail: {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.

I. Introduction

The problem of association rule mining (ARM) in large transactional databases was first introduced in 1993 [3]. 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 [1], not long after the definition of ARM, and was also the subject of much research (see, for example, [1], [8], [27], [10], [14], [20], [4], [11], [26], [16], [6]).

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 such as epidemic control, ailment and treatment pattern discovery, and the detection of medical fraud or misconduct. Similar examples of this imperative are common in credit card companies (international fraud), in the banking industry (international money laundering rings), and elsewhere.

An especially interesting example for large scale distributed databases are peer-to-peer systems. These systems include GRID computing environments such as Condor [17] (20,000 computers), specific area computing systems such as SETI@home [21] (1.8 million computers) or UnitedDevices [25] (2.2 million computers), general purpose peer-to-peer platforms such as Entropia [9] (60,000 peers), and file sharing networks such as Kazaa (5 million peers). Like any other system, large scale distributed systems maintain and produce operational data. However, 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 Kazaa file sharing network. The files shared through Kazaa 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. 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 there can be no global synchronization in a system that large. 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 distant 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 about their local databases with their immediate neighbors 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 speed-ups. 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., [7], [23], [24], [28], 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. 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 [15].

It should be stressed that the goal of our algorithm is not to approximate, but to converge quickly toward the exact solution. This is the same solution that would be reached by a sequential ARM algorithm had all the databases been collected and processed. This convergence can be viewed in two ways: One, that soon after initialization an ever increasing portion of the nodes will compute the exact result. Two, that if the local database changes, or a node disconnects from the system, but the global result remains the same, then while the result of a few nodes might become inaccurate, the correct result will be restored after just a short period and a few message exchanges.

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

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

TABLE I
GENERAL NOTATIONS

Symbol	Meaning
I	Set of items
V_t	Set of nodes at time t
DB_t	The union of all databases at time t
$MinFreq$	Minimal frequency of correct rules
$MinConf$	Minimal confidence of correct rules
$R[DB_t]$	The set of rules which are correct in DB_t
$\tilde{R}_u[DB_t]$	The ad hoc solution of node u at time t

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. We further demonstrate the importance of our majority voting protocol by showing that it can be utilized to solve other problems such as the ranking of rules according to their frequency or confidence. A further generalization allows ranking of rules according to a function of the confidence. For instance, we show how the rules can be sorted according to the Shannon's entropy of the confidence.

No previous protocol was described which discovers the majority locally. However, the majority vote problem is similar to the persistent bit problem, for which local protocols were given ([13], [12]), and the two problems are reducible to one another. The main drawback of the aforementioned persistent bit protocols is that each of them assumes some form of synchronization: In [13], nodes query groups of other nodes and must await a reply before they proceed, while [12] works in locked-step, assuming a global clock pulse. In contrast, our majority vote protocol requires no synchronization at all. There are also more subtle differences which make these protocols impractical for majority vote. For instance, the former only works when the majority is very evident while the latter, because it allows any intermediate result to be corrupted, requires $O(n)$ memory at each node.

II. 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. 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 $Freq(X, DB) = \frac{Support(X,DB)}{|DB|}$. For some frequency threshold *MinFreq* $\in [0, 1]$, 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 *MinConf* $\in [0, 1]$, we say the rule $X \Rightarrow Y$ is *confident* in DB if $Freq(X \cup Y, DB) \geq$ *MinConf* $\cdot\, Freq(X, DB)$. We call confident rules between frequent itemsets *correct* and the remaining rules *false*. The solution of the ARM problem is $R[DB]$ – all the correct rules in the given database.

We assume the database is updated over time, and hence, DB_t will denote the database at time t and $R[DB_t]$ the rules that are correct in that database. In distributed association rule mining the database is partitioned among a set of nodes, and with large-scale distributed mining we allow this set to change over time. Hence we denote V_t the set of nodes at time t. When the number of nodes is large and the frequency of updates is high, it may not be feasible to propagate the changes to the entire system at the rate they occur. Thus, it is beneficial if an incremental algorithm can compute ad hoc results quickly and improve them as more data is propagated. Such algorithms are called *anytime algorithms*.

The performance of an anytime algorithm is measured by its average *recall* and *precision*. Let $\tilde{R}_u[DB_t]$ be the ad hoc solution known to the node u at time t. The recall and precision of u at that time are $\frac{\left|\tilde{R}_u[DB_t] \cap R[DB_t]\right|}{|R[DB_t]|}$ and $\frac{\left|\tilde{R}_u[DB_t] \cap R[DB_t]\right|}{\left|\tilde{R}_u[DB_t]\right|}$. An anytime algorithm is said to be correct if during static periods, in which the database and the system do not change, both the average recall and the average precision converge to one. An important measure of efficiency for an anytime algorithm is the rate of that convergence.

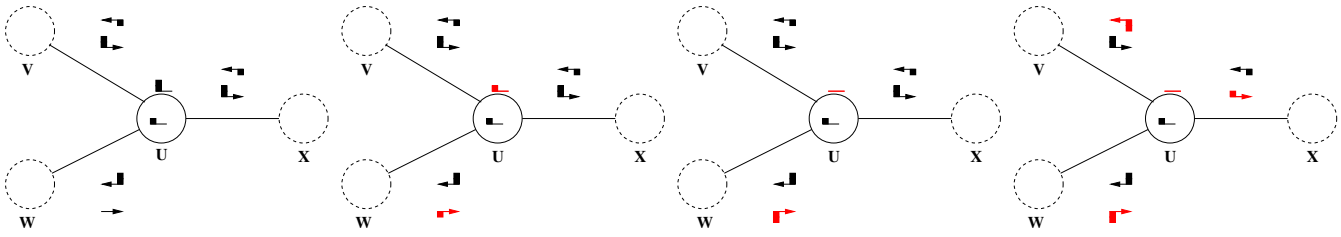
Throughout this work we make two simplifying assumptions. We assume that an underlying mechanism maintains a communication tree that spans all nodes. We further assume the failure model of computers is fail-stop [18], and that a node is informed of changes in the status of adjacent nodes.

III. 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.

A. LSD-Majority Protocol

It has been shown in [19] 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



(a) (b) (c) (d)

Fig. 1. A graphical representation of the LSD-Majority algorithm from the perspective of a single node. Assuming $\lambda = \frac{1}{2}$, we use flagged arrows to symbolize messages to and fro u with the flag height indicating $sum^{vu} - \lambda count^{vu}$. Hence in (a) we see inside the node marked u that $sum^{\perp u} - \lambda count^{\perp u}$ indicates one excess vote and around it the last sent and received messages with different heights. Δ^u, portrayed above u, indicates two excess votes. In (b) a message is received from w which lowers Δ^{wu} and Δ^u but does not trigger additional messages because Δ^u is now equal to Δ^{uv} and to Δ^{ux}. In (c) an additional message from w further lowers Δ^{wu} and Δ^u . Now $\Delta^u < \Delta^{uv}, \Delta^{ux}$ and thus messages are sent to both v and x, as indicated by the change in the height of the flag directed from u to v and to x.

TABLE II
NOTATION FOR LSD-MAJORITY

Symbol	Meaning
$\langle sum^{uv}, count^{uv} \rangle$	Last message sent from u to v
$\langle sum^{vu}, count^{vu} \rangle$	Last message sent from v to u
$\langle sum^{\perp u}, count^{\perp u} \rangle$	$\langle 1,1 \rangle$ if u's input bit is set $\langle 0,1 \rangle$ otherwise
λ	The required majority ratio
E^u	$\{vu : v \text{ and } u \text{ are neighbors}\}$
N^u	$E^u \cup \{\perp u\}$
Δ^u	$\sum_{vu \in N^u} (sum^{uv} - \lambda count^{uv})$
Δ^{uv}	$(sum^{vu} + sum^{uv}) - \lambda (count^{vu} + count^{uv})$

voting protocol – LSD-Majority – which works well for large-scale distributed systems. In the interest of clarity, we assume that the data at each node is a single bit. We will later show how the protocol can 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 agrees with the majority. 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 input 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 – $\langle sum^{uv}, count^{uv} \rangle$ – and the last message it received from v – $\langle sum^{vu}, count^{vu} \rangle$. In the interest of conciseness we symbolize the local bit as a message that is received from $\perp$ and which contains $\langle 1,1 \rangle$if the input bit is set and $\langle 0,1 \rangle$ if the bit is unset. We also extend the group of edges colliding with u, E^u, to include the virtual edge $\perp u$ and designate the extended edge set N^u. Node u calculates the following two functions of these messages and its own local bit:

$$\Delta^u = \sum_{vu \in N^u} (sum^{vu} - \lambda count^{vu})$$

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

Note that if no message was yet received from any neighbor, then Δ^u is positive if the input bit is set and negative if it is unset. Throughout execution the ad hoc output of u is set according to the sign of Δ^u, a majority of set bits if the sign is positive and of unset bits if the sign is negative. Furthermore, Δ^u measures the number of excess set bits u has been informed of (or missing set bits if it is negative). Δ^{uv} measures the number of excess set bits u and v have last reported to one another. Each time the input bit 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.

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 except for the time a message travels from u to v, $\Delta^{uv} = \Delta^{vu}$) and making certain that Δ^{uv} will not mislead v into believing that the number of excess bits 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 have more excess bits then they reported each other; 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, $\left\{ \sum_{wu \neq vu \in N^u} sum^{wu}, \sum_{wu \neq vu \in N^u} count^{wu} \right\}$. Note that after this message is sent, $\Delta^{uv} = \Delta^u$.

Algorithm 1 LSD-Majority

Input for node u: The set of edges that collide with it E^u, An input 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: See notation in table II.

Initialization: For each $vu \in E^u$ set sum^{vu}, $count^{vu}$, sum^{uv}, $count^{uv}$ to 0.

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 $\langle sum, count \rangle$ received over edge vu: Set $sum^{vu} = sum$, $count^{vu} = count$

On any change in Δ^u or Δ^{uv} resulting from a change in the input, 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 $\Delta^{uv} < \Delta^u$ or $\Delta^{uv} \geq 0$ and $\Delta^{uv} > \Delta^u$
Set $sum^{uv} = \sum_{wu \neq vu \in N^u} sum^{wu}$ and $count^{uv} = \sum_{wu \neq vu \in N^u} count^{wu}$
Send $\langle sum^{uv}, count^{uv} \rangle$ over vu to v

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 if 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 of the bits in their combined data cannot be set.

The pseudocode of the LSD-Majority protocol is given in Algorithm 1, and a graphical representation in Figure 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, or $\Delta^v < 0$ for all of them. If there is disagreement 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 the system; 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 given in the Appendix.

B. Verifying Candidate Rule Correctness

A correct rule $X \Rightarrow Y$ ought to satisfy two conditions: the frequency of $X \cup Y$ must exceed $MinFreq$ and the frequency $X \cup Y$ within transactions that contain X must exceed $MinConf$. The first condition can be verified by a majority vote in which each node votes as many times as the number of transactions in its database, with the number of set input bits equal to the support of $X \cup Y$ in that database, and λ set to $MinFreq$. The second condition can be verified by a similar majority vote in which each node votes as many times as the support of X, with the number of set input bits equal again to the support of $X \cup Y$, and λ set to $MinConf$.

Deciding whether a candidate rule is correct or false thus requires two instances of LSD-Majority. A simple optimization would be to share the result of the first instance, which validates the frequency of an itemset among all of the rules derived from that itemset.

The strength of the protocol lies in its behavior when the average of the input bits is somewhat different than the majority threshold λ. Defining the significance of the input as $\frac{1}{\lambda} \frac{\sum_u \left(sum^{\perp u} - \lambda count^{\perp u} \right)}{\sum_u count^{\perp u}}$, we will show in section V-A that even a minor significance, on the scale of ± 0.1, is sufficient for making a correct decision using data from only 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 most of the candidates are far from significant.

C. 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 further candidates should be generated. Second, unlike other algorithms, which produce rules after they have finished discovering all the itemsets, an algorithm which never really finishes discovering all the 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 [5], we periodically consider all the correct rules, regardless of when they were discovered, and attempt to use them for generating new candidates.

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 $\langle r.id, sum, count \rangle$ **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 forever

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

- Gen-Candidates
- For each $r = \langle X \Rightarrow Y \rangle \in C$ and for every $vu \in E^u$
  - If Cond(r,uv) returns true
    - Set $r.sum^{uv} = \sum_{wu \neq vu \in N^u} r.sum^{wu}$ and $r.count^{uv} = \sum_{wu \neq vu \in N^u} r.count^{wu}$
    - Send $\langle r.id, r.sum^{uv}, r.count^{uv} \rangle$ over vu to v

Procedure Cond(r,uv):

Return true on one of the following two conditions:
1. $r.count^{uv} + r.count^{vu} = 0$ and $\Delta^u(r) > 0$
2. $r.count^{uv} + r.count^{vu} > 0$ and either $(\Delta^{uv}(r) \geq 0$ and $\Delta^{uv}(r) > \Delta^u(r))$ or $(\Delta^{uv}(r) < 0$ and $\Delta^{uv}(r) < \Delta^u(r))$

Procedure Gen-Candidates:

1. Set $\tilde{R}_u[DB_t]$ = the set of rules $r = \langle X \Rightarrow Y \rangle \in C$ such that $\Delta^u(r) \geq 0$ and for $r' = \langle \emptyset \Rightarrow X \cup Y \rangle$ $\Delta^u(r') \geq 0$
2. For every $r = \langle X \Rightarrow Y \rangle \in \tilde{R}_u[DB_t]$, such that $X = \emptyset$ and $i \in Y$ if $r' = \langle Y \setminus \{i\} \Rightarrow \{i\} \rangle \notin C$ insert r' into C with $r'.sum = r'.count = 0$, $r'.\lambda = MinConf$ and $r'.id =$ unique rule id
3. For each $c_1 = \langle X \Rightarrow Y \cup \{i_1\} \rangle, c_2 = \langle X \Rightarrow Y \cup \{i_2\} \rangle \in \tilde{R}_u[DB_t]$ such that $i_1 < i_2$, if $c_3 = \langle X \Rightarrow Y \cup \{i_1, i_2\} \rangle \notin C$ and $\forall i_3 \in Y$: $\langle X \Rightarrow Y \cup \{i_1, i_2\} \setminus \{i_3\} \rangle \in \tilde{R}_u[DB_t]$, add c_3 to C with $c_3.sum = c_3.count = 0$, $c_3.\lambda = c_1.\lambda$, and $c_3.id =$ unique rule id

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 [2]. 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 verifying 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 that must be generated by any deterministic algorithm.

In the case of Majority-Rule, the dynamic nature of the 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, $\tilde{R}_u[DB_t]$ 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 $\emptyset \Rightarrow X \in \tilde{R}_u[DB_t]$, 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 $\tilde{R}_u[DB_t]$ 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 $\tilde{R}_u[DB_t]$ contains only correct rules, then no superfluous candidate rules are ever generated using this method. Note that when this method is used, generating a candidate of the form $X \Rightarrow Y$ always generates the matching candidate $\emptyset \Rightarrow X \cup Y$.

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 a rule identifier – $r.id$ – in addition to sum and $count$. 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. Note that although the algorithm as given runs in an infinite loop, this is strictly for the purpose of clarity. It is straightforward to implement it in an event-based manner. Hence, when there are no new candidates, there is no need to read further transactions and all communication is in response to incoming messages, just as in LSD-Majority.

IV. Generalized Majority Voting

The previous section we described a majority voting algorithm scalable for peer-to-peer networks and an application of that algorithm for association rule mining. In this section we will present additional applications of the majority voting algorithm. Specifically, we will show that given multiple options voting, with each peer u choosing one or more of d options, the different options can be ranked according to their total popularity; i.e., $Rank\left(\sum_u support_i^u\right)$ can be computed for every option i. Computing the rank of options is equivalent to *ordering* them, which means that by computing the ranks we can come up with a local method of ordering rules according to their confidence or their support.

Another application of the majority voting protocol is for cases in which the rank should be computed not on $\sum_u x_i^u$, but rather on some function $\mathcal{F}$ of the sum. This is straightforward if $\mathcal{F}$ is monotone, because the same i which maximizes $\sum_u x_i^u$ would maximize either $\mathcal{F}\left(\sum_u x_i^u\right)$ or $-\mathcal{F}\left(\sum_u x_i^u\right)$. However, we show that the rank of $\mathcal{F}\left(\sum_u x_i^u\right)$ can be computed for a group of piecewise monotone functions, which include functions such as Shannon's Entropy $\mathcal{H}$ which are often used for rule ranking [22].

A. Multiple Choice Voting

Suppose a group of peers would like to agree on one of d options. Each peer u conveys its preference by initializing a binary preference vector $P^u \in \{0,1\}^d$, setting the preferred option to one and the rest to zero (alternatively, several options can be set to one with no change in the algorithm). To decide which option has the largest support we will transform each vector to a d by d matrix A^u such that $A^u[i,j] = P^u[i] - P^u[j] + 1$. Now, for each entry of A^u we can perform an independent instance of LSD-Majority, with $sum^{\perp u} = A^u[i,j]$, $count^{\perp u} = 2$ and $\lambda = 0.5$.

Algorithm 3 Rank-By-Support

Input for node u: The set of edges that collide with it E^u, a list of d association rules S with their respective local support counts P^u.

Output: The algorithm never terminates. Nevertheless, at each time the function $Rank(i)$ can be called which returns the rank of the i^{th} rule.

Initialization:

Compute $A^u[i,j] = P^u[i] - P^u[j] + M$, where $M = \max\{P^u\} - \min\{P^u\}$.

For each pair $i,j \in [1,d]$ initialize LSD-Majority with the following inputs: $sum^{\perp u} = A^u[i,j]$, $count^{\perp u} = 2M$, $\lambda = 0.5$ and with the variables $\Delta^u_{A^u[i,j]}$ and $\Delta^{uv}_{A^u[i,j]}$ for each $v \in E^u$.

To evaluate $Rank(i)$:

Count the number r of indexes j such that $\Delta^u_{A^u[i,j]} < 0$ and return $r+1$.

Theorem 1: The result of LSD-Majority performed on the $[i,j]$ entry of A^u of all $u \in V$ is that $\Delta^u \geq 0$ if and only if $\sum_u P^u[i] \geq \sum_u P^u[j]$

Proof: Follows immediately from the correctness of LSD-Majority and from the initialization of A^u. ■

Corollary 1: Let $J_i = \{j_1, \ldots, j_M\}$ be the group of indexes such that for all $j \in J_i$ the result of LSD-Majority on the $[i,j]$ entry of A^u is that $\Delta^u < 0$ and for all $j' \notin J_i$ the result is that $\Delta^u \geq 0$ then $\sum_u P^u[i]$ is the $M+1$ most supported option.

Corollary 1 gives us a simple way of computing the order of a set of d rules according to their support or confidence. All we have to do is perform d^2 local majority votes and return for each option the size of J_i. The pseudocode of this algorithm is given in Algorithm 3.

B. Ranking Functions of the Support

Ordering rules according to their support or confidence may sometime be simplistic. A large body of work exists on different functions of those arguments which can result in better ordering of rules. Next we would outline how such functions can be computed locally. The simplest example is computing the rank of any monotone function of the support, $\mathcal{F}_\mathcal{M}\left(\sum_u P^u[i]\right)$. Assuming that $\mathcal{F}_\mathcal{M}$ is monotonously increasing, the ranking of $\sum_u P^u[i]$ dictates the ranking of $\mathcal{F}_\mathcal{M}\left(\sum_u P^u[i]\right)$. For monotonously decreasing functions the same computation is made and then the complement of the result to d is returned.

An interesting generalization of the problem is when the function F is piece-wise monotone with symmetric pieces. Such functions include trigonometric functions (e.g., $\sin$), the Shannon's Entropy $\mathcal{H}$ (considering the average frequency as an estimator of the mean of a Bernoulli random variable), certain polynomials (e.g., x^2), and more.

To see how this can be done suppose the input of each peer u is a binary vector P^u sampled from d random variables distributed according to Bernoulli distributions such that $P^u[1] \sim Ber(p_1), \ldots, P^u[d] \sim Ber(p_d)$. The average input vector, $\bar{P}[1], \ldots, \bar{P}[d]$ is an estimator of $p_1, \ldots, p_d$. The Shannon's entropy of each of the random variables, $\mathcal{H}(p)$ is defined on the range $[0, 1]$, receives its minimal values (zero) on $p = 0$ and $p = 1$, its maximal value (one) on $p = 0.5$, and is symmetric around $p = 0.5$. Now suppose we want to rank the variables according to their $\mathcal{H}$ value. Since $\mathcal{H}(p)$ is monotonously increasing for $p \in [0, 0.5]$ and monotonously decreasing for $p \in [0.5, 1]$ a straightforward implementation of the ranking algorithm would solve the problem had all the p_i been in one of these ranges. Putting $q_i = 1 - p_i$ it is obvious that for all i either p_i or $q_i$are in $[0, 0.5]$.

The main idea of the algorithm is to double the size of the vector by considering $1 - P^u[i]$ as well for each i, and then rank the entries of the extended vector. Concurrently, for each i we will decide using LSD-Majority whether $\bar{P}[i] < 0.5$. Now, we choose to compare p_i or q_i to p_j or q_j according to which of them are estimated smaller than 0.5. Assuming, for example, that $p_i < 0.5$ and $q_j < 0.5$, $\mathcal{H}(p_i)$ would be larger than $\mathcal{H}(p_j)$ if and only if $p_i > q_j$. Generally, renaming $x_i = \begin{cases} p_i & p_i < 0.5 \\ q_i & otherwise \end{cases}$ we can write the following Theorem:

Theorem 2: $\mathcal{H}(p_i) \geq \mathcal{H}(p_j)$ if and only if $x_i \geq x_j$.

To translate Theorem 2 into an algorithm (Algorithm 4) we first notice that $p_i - p_j = q_j - q_i$. Thus, the number of comparisons can be reduced by half. Our algorithm will compare just p_i and q_i to p_j ($2d^2$ comparisons rather than $4d^2$) and p_i to 0.5 (a further d comparisons). We initialize for every P^u two $d \times d$ matrices A^u and B^u. Where $A^u[i,j] = P^u[i] - P^u[j] + 1$ and $B^u[i,j] = (1 - P^u[i]) - P^u[j] + 1$. Now, for every entry of A^u and B^u we perform LSD-Majority with $sum^{\perp u}$ set to $A^u[i,j]$ or $B^u[i,j]$ respectively, $count^{\perp u}$ set to two and $\lambda = 0.5$. Additionally, for every entry of P^u we perform LSD-Majority with $sum^{\perp u} = P^u[i]$, $count^{\perp u} = 1$ and $\lambda = 0.5$.

Let, $\Delta^u_{P^u[i]}$, $\Delta^u_{P^u[j]}$, $\Delta^u_{A^u[i,j]}$, $\Delta^u_{B^u[i,j]}$ be the results of LSD-Majority for the i and j entries of P^u and for the $[i,j]$ entries of A^u and B^u. To decide whether $\mathcal{H}(p_i) \geq \mathcal{H}(p_j)$ we first check $\Delta^u_{P^u[i]}$ and $\Delta^u_{P^u[j]}$ are smaller than 0. If so, then according to Theorem 2, $\mathcal{H}(p_i) \geq \mathcal{H}(p_j)$ if and only if $\Delta^u_{A^u[i,j]} \geq 0$. Similarly, if $\Delta^u_{P^u[i]} \geq 0$ and $\Delta^u_{P^u[j]} \geq 0$ then $\mathcal{H}(p_i) \geq \mathcal{H}(p_j)$ if and only if $\Delta^u_{A^u[i,j]} < 0$. The cases for $\Delta^u_{P^u[i]} \geq 0$ and $\Delta^u_{P^u[j]} < 0$ or $\Delta^u_{P^u[i]} < 0$ and $\Delta^u_{P^u[j]} \geq 0$ are likewise computed using $\Delta^u_{B^u[i,j]}$.

Algorithm 4 Rank-By-$\mathcal{H}$

Input of node u: The set of edges that collide with it E^u, a binary vector P^u of length d.
Output of node u: The algorithm never terminates. Nevertheless, at each time it computes a function $J : [1,k] \rightarrow [1,k]$ which is the rank of the Shannon's Entropy of each entry of the vector.

Initialization:

Compute $A^u[i,j] = P^u[i] - P^u[j] + 1$,
$A^u[i,j] = (1 - P^u[i]) - P^u[j] + 1$.

For each entry of A^ustart LSD-Majority with the following inputs: $sum^{\perp u} = A^u[i,j]$, $count^{\perp u} = 2$, $\lambda = 0.5$ and with the variables $\Delta^u_{A^u[i,j]}$ and $\Delta^{uv}_{A^u[i,j]}$ for each $v \in E^u$; start similar instances for B^u.

For each entry of P^u start LSD-Majority with the following inputs: $sum^{\perp u} = P^u[i]$, $count^{\perp u} = 1$, $\lambda = 0.5$ and with the variables $\Delta^u_{P^u[i]}$ and $\Delta^{uv}_{P^u[i]}$ for each $v \in E^u$.

To evaluate $J(i)$**:**

Count the number M of indexes j for which condition $Cond(i,j)$ holds true, and return $M + 1$.

$Cond(i,j)$: return true if one of the following conditions holds:

$$\left\{ \begin{array}{l} \Delta^u_{P^u[i]} < 0 \text{ and } \Delta^u_{P^u[j]} < 0 \text{ and } \Delta^u_{A^u[i,j]} \geq 0 \\ \Delta^u_{P^u[i]} < 0 \text{ and } \Delta^u_{P^u[j]} \geq 0 \text{ and } \Delta^u_{B^u[i,j]} < 0 \\ \Delta^u_{P^u[i]} \geq 0 \text{ and } \Delta^u_{P^u[j]} < 0 \text{ and } \Delta^u_{B^u[i,j]} \geq 0 \\ \Delta^u_{P^u[i]} \geq 0 \text{ and } \Delta^u_{P^u[j]} \geq 0 \text{ and } \Delta^u_{A^u[i,j]} < 0 \end{array} \right\}$$

V. 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 lock-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.

For lack of real datasets of the magnitude required by a system of 1600 computers, we used synthetic databases generated by the standard tool from the IBM-quest data mining group [2]. 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

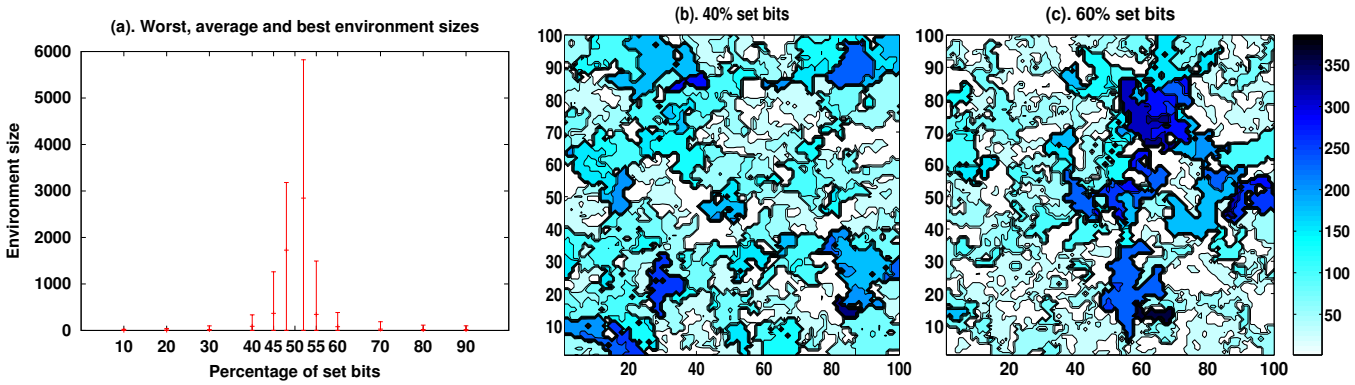


Fig. 2. The locality of LSD-Majority strongly depends on the difference between the percentage of set bits in the data and the threshold. We ran LSD-Majority on a network of 10,000 nodes which were randomly initialized with different percentages of set bits. The leftmost figure shows the maximum, average and minimum of the environment size over the 10,000 nodes. If the percentage of ones is more than 60 or less than 40, then the average environment size is dozens of nodes or less. As the percentage approaches the threshold, more data must be gathered in order to reach the decision (all the data is needed by most nodes if the average is 50%). The remaining figures are contour maps showing the environment sizes of nodes in different areas of the grid. What is interesting about these maps is that there is no clear pattern. Had there been a pattern (for example, were environments nearer to the center larger than at the rim), it would have indicated that the protocol distributes the work unfairly.

average pattern length. The combined size of each of the three databases is 10,000,000 transactions.

With the default settings, the number of patterns generated is $10,000$, which results in a very high proportion of false rules vs. correct ones: about ten thousand candidates are required for each candidate which turns out to be correct. Since we simulated thousands of computers, using the default number of patterns would pose impracticable memory requirements on our simulator. Since we can show that Majority-Rule performs much better for false rules than for correct ones, we could reduce the proportion of the former without impairing the validity of our experiments. Hence we changed this parameter to one hundred in our simulations, resulting in about hundred false candidates per correct rule.

The main argument which would dictate the performance of Majority-Rule on a real-life database is the distribution of the significance of candidates in that database. It has already been shown that this distribution can vary from one database to another and further depends on the choice of $MinFreq$ and $MinConf$. Since we thoroughly investigate the behavior of the majority vote with respect to the significance of the input, and since different instances of the vote (i.e., different candidates) map into independent votes, the expected performance for each real-life database can be projected according to the distribution of candidate frequency in that database.

A. Locality of LSD-Majority and Majority-Rule

In a static system, we define the environment of u as the nodes whose bits gets counted by u. The locality of a protocol is measured according to the worst-case size and the average size of the environments of the different nodes. Since the number of nodes in LSD-ARM is very large, any algorithm which does not have good average locality simply cannot be considered scalable or practical. Locality is also instrumental to performance. Since no messages sent outside the environment of u will ever be propagated to u, the size of the average environment determines the number of messages exchanged during the protocol and the number of steps required in a lock-step execution. A protocol that has good locality will thus also be quick and communication efficient.

In LSD-Majority, the size of the environment of a node is simply the number of data bits it has received by the time the simulation terminates (i.e., no further messages are exchanged). It is thus equal to $1 + \sum_{vu \in E^u} count^{vu}$.

Our experiments with LSD-Majority show that its locality strongly depends on the significance of the input: $\frac{1}{\lambda} \frac{\sum_u \left(sum^{\perp u} - \lambda count^{\perp u}\right)}{\sum_u count^{\perp u}}$. Figure 2 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 messages at all.

In figure 3 we show what happens when the distribution of the data across the grid is biased. In subfigure 3(a), the distribution of set input bits has a left-to-right gradient: the left side of the grid has five percent fewer set input bits

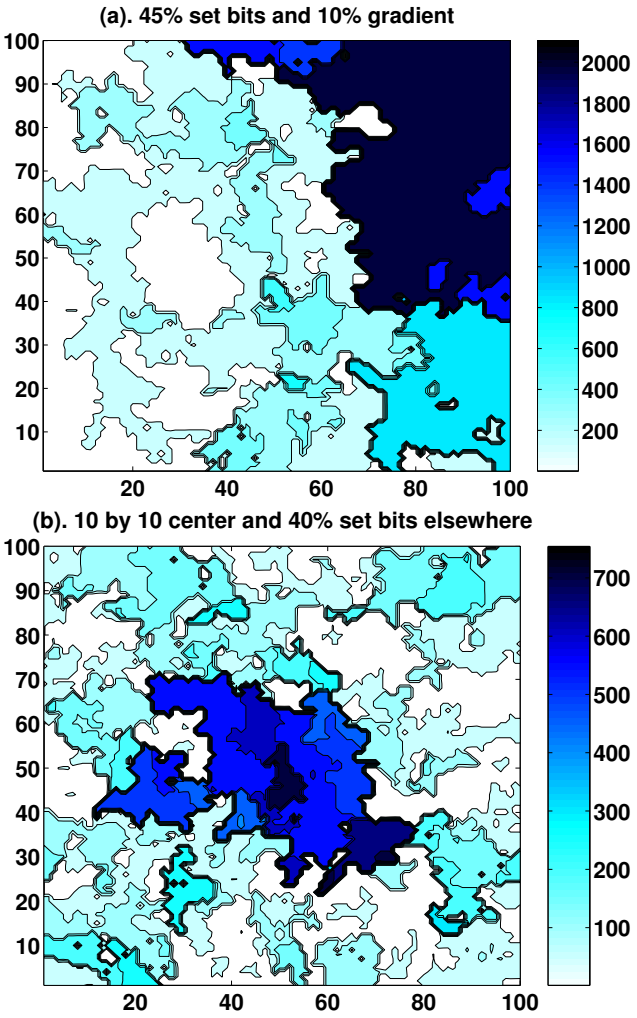


Fig. 3. LSD-Majority remains local even if the distribution is biased. In the left figure, the right-hand side has 50% set input bits and the left-hand side 40%; in the figures on the right, the center hundred nodes are all set bits and in the rest 40% of the bits are set. The size of each node's environment depends on the average of bits in close proximity to it. See for reference figure 2 (b) which has the same percentage of set input bits as the left side of subfigure (a) and the rim of subfigure (b), and indeed has a similar distribution of environment sizes.

and the right side five percent more set input bits than the average. In subfigure 3(b), the input bits of the hundred central nodes are all set, and the rest were randomly selected. The results show that the locality characteristics of each area of the grid depend on its small range average. For instance, the left side of subfigure 3(a) and the rim of subfigure 3(b), both of which have 40% set input bits, are very similar to one another and to figure 2(b), which has the same percentage of set input bits across the whole grid. The outcome of these experiments is that LSD-Majority remains local even if the distribution is strongly biased.

For Majority-Rule we define the environment size of a candidate r in node u to be the number of transactions which u was informed of, in messages related to candidate r, until the algorithm reached consensus. Figure 4 presents the locality of Majority-Rule. For each rule the worst-case and the average environment sizes are drawn against that rule's global significance. As can be seen, the decrease in the environment size as the significance grows away from 0 is even stronger than that seen in the stand-alone LSD-Majority experiments. This is because the local frequency and confidence of a rule only becomes static after all the transactions in the database are scanned. With the databases and the M we used, this took 60 steps for each rule at each node. Apparently, for rules with extreme significance only the first few updates were communicated, after which the protocol converged.

B. 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 discovered – and precision – the percentage of correct rules in the ad hoc solution – vis-a-vis the number of transactions scanned. Figure 5 describes the convergence of the recall (a) and of the precision (b). At the bottom of this figure 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 convince

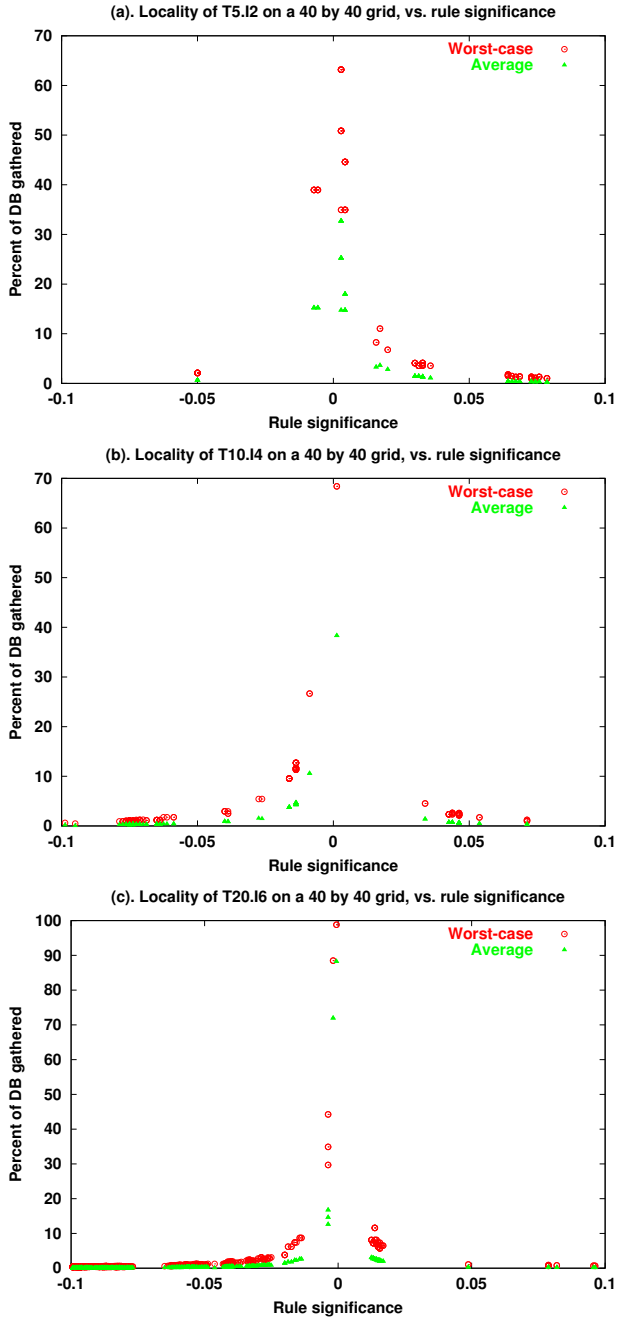


Fig. 4. Locality of Majority-Rule. Figures 4(a) through 4(c) give, for T5.I2, T10.I4 and T20.I6, the environment size, in percentage of the global database that is collected at the average node vs. the significance of the rule. All of these figures are very similar to figure 2(a). That is, despite the fact that LSD-Majority is a simplification of Majority-Rule, they have similar locality characteristics.

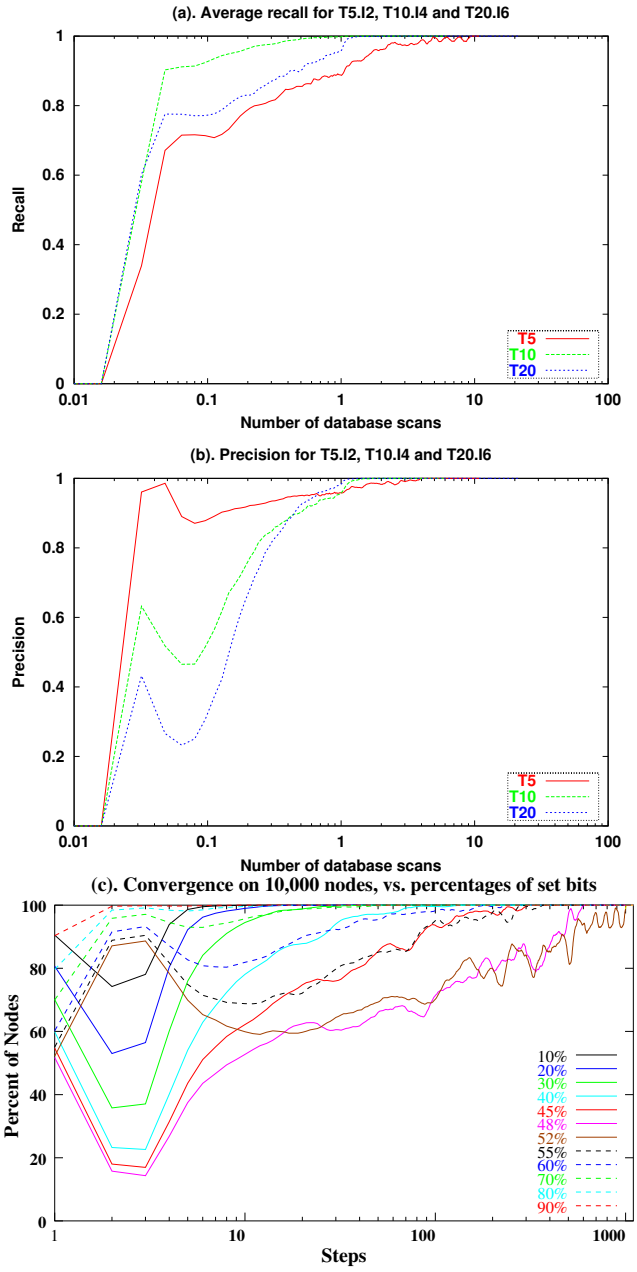


Fig. 5. Subfigure (a) describes the convergence of the recall of Majority-Rule at the average node, for three different databases. Similarly, subfigure (b) describes the convergence of the precision. By the end of the first (parallel) database scan, both the the recall and the precision of the average node are above 90 percent. The bottom figure describes the convergence of LSD-Majority – the percent of the nodes which calculate the correct majority, for different percentages of set input bits. Starting from about step 10, the distance between pairs of lines is nearly the same. In other words, the convergence rate has exponential dependency on significance.

nodes that the majority is 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).

We have two objectives where communication load is concerned. One is that the load be small and the other is that it be evenly distributed on the grid. The first requirement is important for nondedicated systems, in which applications compete over bandwidth. The second requirement embodies the desire for fairness – that the nodes should do approximately the same work – which is essential in peer-to-peer systems. Figure 6 shows the distribution of the communication load vis-a-vis rule significance. For rules that are very near the threshold, a lot of communication[2] 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 hardly any messages at all.

C. Majority-Rule in Dynamic Scenarios

The local frequency and confidence of each rule changes several times from the time the candidate is generated and until it is sought in all the transactions of the local database. So each execution of Majority-Rule is, in fact, data-dynamic. To further investigate dynamic scenarios, we performed controlled experiments with LSD-Majority. We experimented with two different models, both of which are stationary (i.e., while the data changes, the result stays the same): a steady-state model, in which the data at the nodes is constantly changing, and an abrupt change model in which the topology changes and then the system remains static until the protocol converges.

Under the steady state model, we investigate the noise immunity of the protocol. In this experiment we randomly select, once every step, one percent of the nodes, and flip them from zero to one or vice versa (making sure the overall percentage of ones remains the same). Figure 7(a) describes the percentage of nodes which concluded, at each step, that the majority is of set bits, for various percentages of bits that were actually set. As the graph shows, when the percentage of nodes which had their bit set was 70 and above or 30 and below, more than 95 percent of the nodes computed the correct majority. This means that when the majority is clear, the noise immunity of LSD-Majority is above 95 percent. The noise immunity of Majority-Rule will, naturally, depend on the proportion of significant vs. insignificant rules. About 200 messages are sent each step for 10% set bits and 90% set bits (only twice the number of nodes flipped), 300 for 20% and 80%, and 500 for 30% and 70%.

In the abrupt change model, we investigated the response of the protocol in the event that, after it converged, 10 random edges were disconnected (simulating the failure of random nodes). From figure 8(a) it can be seen that the protocol reconverged in just a few dozen steps (depending on the significance of the data), and that at its peak, no more than a few percent of the nodes were led to the wrong result. The effect of an abrupt change on communications is also moderate. Naturally it lasts the same number of steps. At its peak, fewer than one hundred messages are sent by the 10,000 nodes.

[2]It is important to keep in mind that we consider here each pair of integers we send a message. In a realistic scenario, a message will contain up to 1500 bytes, or about 180 integer pairs.

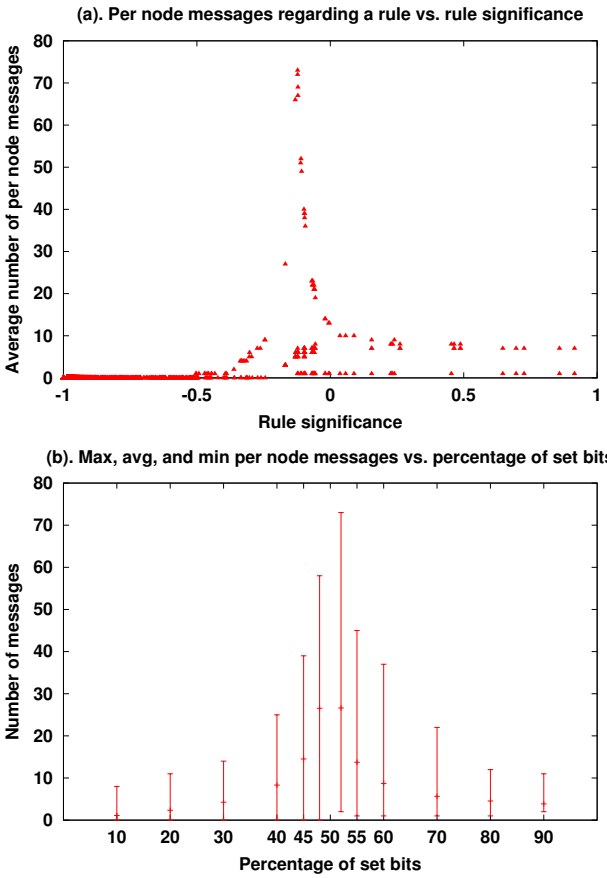


Fig. 6. Communication characteristics of Majority-Rule (a), and of LSD-Majority (b). Note that a message here is a pair of integers. In a realistic scenario each message would contain more than one hundred such pairs, so the costs would be amortized.

VI. 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

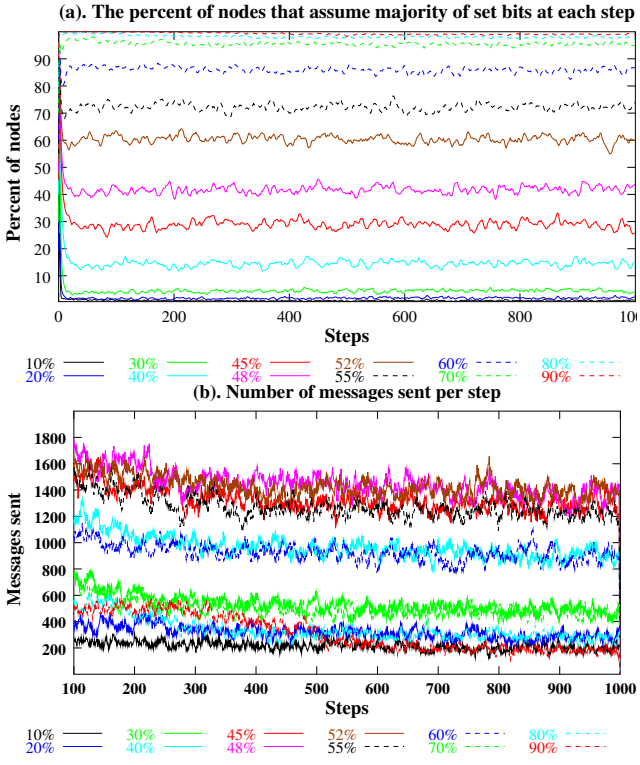


Fig. 7. LSD-Majority under the steady-state model. The noise immunity of LSD-Majority depends on the significance of the majority. For a significant majority, ± 0.4, more than 95 percent of the nodes retain the correct result, for significance of ± 0.2 about 85 percent, and for ± 0.1 about 70 percent. The communication required to retain the correct result is, not surprisingly, dependent on the percentage of the nodes that are wrong, and hence, of the significance of the majority.

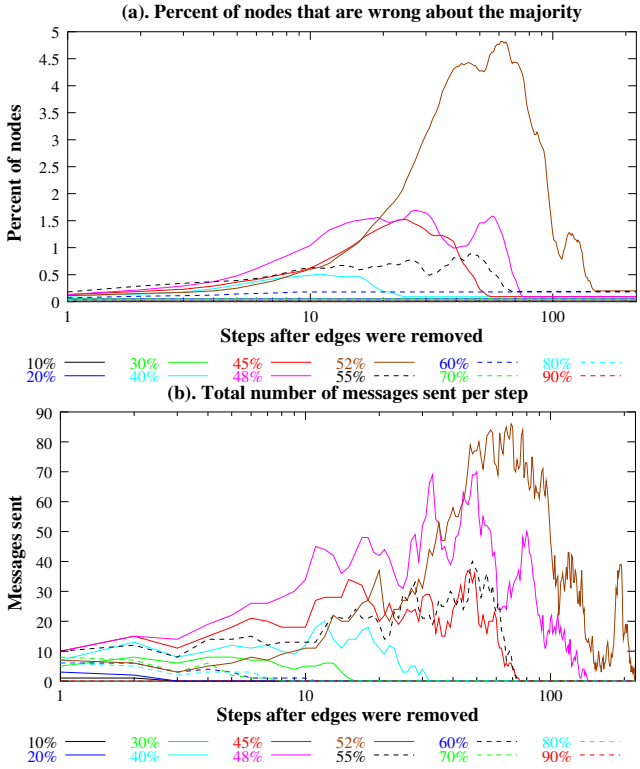


Fig. 8. LSD-Majority under the abrupt change model. The significance of the majority dictates both the number of steps and the number of messages required for reconvergence. Nevertheless, at no step do more than a few percent of the nodes err about the correct majority, nor do more than one percent of the nodes send messages to one another.

the overwhelming majority of the candidates usually turn out this way, the communication load of Majority-Rule depends mainly on the size of the output – the number of correct rules. That number is controllable via user supplied parameters, namely $MinFreq$ and $MinConf$.

We have shown that our algorithm functions well even if the data in the distributed machines constantly changes, and that the communication load which results from these changes is minor. This makes Majority-Rule especially well-suited for incremental mining and for applications which require the ongoing monitoring of a large-scale distributed system. Such applications exist in the areas of intrusion detection and network traffic control.

References

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

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

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Int'l. Conference on Management of Data*, pages 207–216, Washington, D.C., June 1993.

[4] V. S. Ananthanarayana, D. K. Subramanian, and M. Narasimha Murty. Scalable, distributed and dynamic mining of association rules. In *Proc. of the Int'l Conference on High Performance Computing HiPC'00*, pages 559–566, 2000.

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

[6] D. Cheung and Y. Xiao. Effect of data skewness in parallel mining of association rules. In *12th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 48 – 60, April 1998.

[7] David Wai-Lok Cheung, Jiawei Han, Vincent Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of 12th International Conference on Data Engineering, ICDE*, pages 106–114, 1996.

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

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

[10] Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):352 – 377, 2000.

[11] P. Iko and M. Kitsuregawa. Parallel fp-growth on pc cluster. In *Seventh Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD03)*, 2003.

[12] S. Kutten and B. Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220(1):93–111, 1999.

[13] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proc. of the ACM Symposium on Principle of Distributed Computing (PODC)*, pages 20–27, Ottawa, Canada, August 1995.

[14] Jun-Lin 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.
[15] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. of the Int'l Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.
[16] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *Proc. of ACM Int'l. Conference on Information and Knowledge Management*, pages 31 – 36, Baltimore, MD, November 1995.
[17] The Condor Project. `http://www.cs.wisc.edu/condor/`.
[18] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.
[19] A. Schuster and R. Wolff. Communication-efficient distributed mining of association rules. In *Proc. of the ACM SIGMOD Int'l. Conference on Management of Data*, pages 473 – 484, Santa Barbara, California, May 2001.
[20] A. Schuster, R. Wolff, and D. Trock. A high-performance algorithm for distributed association rule mining, 2003.
[21] Seti@home. `http://setiathome.ssl.berkeley.edu/`.
[22] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right objective measure for association analysis. *Information Systems*, 29(4):293 – 313, June 2004.
[23] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. of the ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, pages 263 – 266, Newport Beach, CA USA, August 1997.
[24] Shiby Thomas and Sharma Chakravarthy. Incremental mining of constrained associations. In *Proc. of the Int'l Conference on High Performance Computing HiPC'00*, pages 547–558, 2000.
[25] United devices inc. http://www.ud.com/home.htm.
[26] Osmar R. Zaiane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rules mining without candidacy generation. In *IEEE 2001 International Conference on Data Mining (ICDM'2001)*, pages 665–668, 2001.
[27] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.
[28] Minghua Zhang, Ben Kao, David Wai-Lok Cheung, and Chi Lap Yip. Efficient algorithms for incremental update of frequent sequences. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 186–197, 2002.

APPENDIX

A simple proof that if the LSD-Majority algorithm converges (i.e., no further messages should be sent) then all nodes converge to the same majority value is given in Section III. Here we will prove that all of the nodes converge to the *correct* majority, i.e., that upon convergence $\Delta^u \geq 0$ if and only if the majority of the nodes which can be reached from u have their input bit set.

Assume a static connected tree $G(V,E)$ with static data $\langle sum^{\perp u}, count^{\perp u} \rangle$ at each $u \in V$. Let $count^{uv}$ and sum^{uv} be the values after the algorithm has converged. Let Δ^u, Δ^{uv} be as defined in Algorithm 1. We say an edge uv is *unemployed* if no message was sent over that edge during the execution of the algorithm. For each $u \in V$ let $[u] = \{v \in V : v \text{ is reachable from } u \text{ using edges in } E\}$. For every $uv \in E$ let $[u]_v = \{w \in V : w \text{ is reachable from } v \text{ using edges in } E \setminus \{uv\}\}$. Finally, for any subset of nodes $S \subseteq V$ let $\Delta_S = \sum_{v \in S} sum^{\perp v} - \lambda count^{\perp v}$.

Theorem 3: In a static tree $G(V,E)$ with static data and upon convergence, for all $u \in V$, $\Delta^u \geq 0$ if and only if $\Delta_V \geq 0$

Lemma 1: In a static tree $G(V,E)$ with static data and upon termination if for some u $\Delta^u \geq 0$ then for each v such that $uv \in E$, $sum^{vu} - \lambda count^{vu} \leq \Delta_{[u]_v}$

Proof: By induction on $|[u]_v|$

Base: $|[u]_v| = 1$ which means that $[u]_v = \{v\}$ and v received no messages from nodes other than u and sent no messages to nodes other than u, i.e., any edge $wv \neq uv$ is unemployed. Hence, $\Delta_{[u]_v} = sum^{\perp v} - \lambda count^{\perp v}$ and $\Delta^v = sum^{uv} + sum^{\perp v} - \lambda\left(count^{uv} + count^{\perp v}\right)$. We assume, by contradiction, $sum^{vu} - \lambda count^{vu} > \Delta_{[u]_v} = sum^{\perp v} - \lambda count^{\perp v}$. It follows that $\Delta^{vu} = \Delta^{uv} = sum^{vu} + sum^{uv} - \lambda\left(count^{vu} + count^{uv}\right) > sum^{\perp v} + sum^{uv} - \lambda\left(count^{\perp v} + count^{uv}\right) = \Delta^v$. Hence either $\Delta^{uv} < 0$, in which case u should send a message to v, or $\Delta^{vu} \geq 0$ in which case v should send a message to u. Either case is contradictive to the convergence assumption.

Step: Assuming the lemma holds for $|[u]_v| \leq k$, we will prove that it holds for $|[u]_v| = k+1$. From the induction hypothesis we learn that for each edge vw other than vu (there must be at least one, or $|[u]_v| = 1$), $sum^{wv} - \lambda count^{wv} \leq \Delta_{[v]_w}$.

$$\begin{aligned}
\Delta^v &= sum^{\perp v} - \lambda count^{\perp v} + \sum_{wv \in E} sum^{wv} - \lambda count^{wv} \\
&\quad \text{because of the induction hypothesis} \\
&\leq sum^{uv} - \lambda count^{uv} + sum^{\perp v} - \lambda count^{\perp v} + \sum_{wv \neq uv \in E} \Delta_{[v]_w} \\
&= sum^{uv} - \lambda count^{uv} + \sum_{x \in [v]_u} sum^{\perp x} - \lambda count^{\perp x} \\
&\quad \text{the contradictive assumption} \\
&\leq sum^{uv} - \lambda count^{uv} + sum^{vu} - \lambda count^{vu} \\
&= \Delta^{uv}
\end{aligned}$$

Again, either $\Delta^{uv} < 0$, in which case u should send a message to v or $\Delta^{vu} \geq 0$ in which case v should send a message to u, and both cases are contradictory to the convergence assumption. ■

Corollary 2: If $\Delta^u < 0$ then $sum^{vu} - \lambda count^{vu} \geq \Delta_{[v]_u}$

Proof: We now prove Theorem 3. First, recall that if node u has $\Delta^u \geq 0$ and no node $v \in V$ needs to send a message, then every node v must have $\Delta^v \geq 0$. We now must prove that $\Delta^u \geq 0$ if and only if $\Delta_V \geq 0$. To do so, we add a fictitious node f with $c_f = s_f = 0$ to an arbitrary node u. Note that if u does not need to send a message to f then $\Delta^f = \Delta^{uf} = \Delta^u$, and also that f never sends messages, and hence does not affect the protocol in any way. Upon convergence, according to the lemma, $0 \leq \Delta^f = \Delta^{uf} \leq \Delta_{[f]_u} = \Delta_V$. Since we know $\Delta^f \geq 0$, Δ_V is also greater than or equal to 0. ■