A distributed recommender system architecture

Abstract. In contemporary internet architectures, including server farms and blog aggregators, web log data may be scattered among multiple cooperating peers. In order to perform content personalization through provision of recommendations on such architectures, it is necessary to employ a recommendation algorithm; however the majority of such algorithms are centralized, necessitating excessive data transfers and exhibiting performance issues when the number of users or the volume of data increase. In this paper we propose an approach where the clickstream information is distributed to a number of peers, which cooperate for discovering frequent patterns and for generating recommendations, introducing (a) architectures that allow the distribution of both the content and the clickstream database to the participating peers and (b) algorithms that allow collaborative decisions on the recommendations to the users, in the presence of scattered log information. The proposed approach may be employed in various domains, including digital libraries, social data, server farms and content distribution networks.

Keywords: distributed recommender system, frequent-pattern mining, generalized association rule mining, FP-Growth, GP-Close, FGP, load balancing

1 Introduction

The increasing popularity of *social media* applications has brought our attention to the need to *personalize* their content, in order for users to easily locate articles of interest, which can be performed, either via *collaborative filtering*, or *frequent-pattern mining*. However, the traditional web 2.0 recommendation-offering applications, such as *blog aggregators*, are centralized, which results in decreased performance, since a single server reads the provided feeds and is thus responsible for making the recommendations. Furthermore, we have to consider that in the typical case of a number of web 2.0 applications, both the published material and the clickstream information are scattered on multiple servers, and therefore a centralized model for our recommender system does not fit well with the existing deployment paradigm. Finally, the single-server approach exhibits *limited scalability* when the number of users or the volume of data substantially increase, necessitating the adoption of distributed schemes. However, the majority of the existing distributed recommendation algorithms assume the existence of a centralized transaction database, which is *partitioned* among the involved processors, while only a few of them are able to be implemented on a *shared-nothing* application scenario, and can thus be executed on more than one processors independently (Pramudiono, et al., 2005; Li et al., 2008). The above issues show the way for the *peer-to-peer* (P2P) paradigm, which -besides being distributed- matches the architecture of federated digital libraries, content distribution networks, blog aggregators and server farms, where participating content providers are independent of each other and no particular hierarchy is imposed on them.

In a recent work, Giannikopoulos et al. (2008) have combined FP-Growth (Han et al., 2004) a well-known frequent-pattern mining algorithm, with GP-Close (Jiang and Tan, 2006; Jiang et al., 2007), which performs generalized association rule mining (GARM), into an efficient approach called *FGP*, incorporating the hierarchy information present in a taxonomy graph, exploiting the fact that the tags annotating a post are typically hierarchically organized. An extension of FGP, called FGP+, where a node is allowed to have multiple parents within the taxonomy, has been proposed in (Giannikopoulos et al., 2010).

Nevertheless, the FGP algorithm was applied to a centralized web access log, which exhibited performance shortcomings; therefore, it would be better to utilize a P2P system, in order to provide a *distributed recommendation engine*, so as to apply FGP to each peer separately.

In brief, the contributions of our work are outlined in what follows:

- We propose a novel recommendation algorithm for the P2P domain, which personalizes the users' browsing experience, taking into account the clickstream information present in the (distributed) web access logs.
- We outline the architecture of a distributed recommendation system, which overcomes the performance bottleneck of current centralized applications.
- We introduce a distributed frequent-pattern mining algorithm, which takes into account the taxonomic information inherent in social media applications, server farms, as well as content distribution networks.

The rest of the paper is organized as follows: In section 2, we provide an overview of related research in the fields of centralized and distributed association rule mining (forming the basis of frequent-pattern mining and GARM) for web personalization, as well as P2P systems. In section 3, the architecture of our distributed recommendation system is described. In the following section, we focus on the experimental evaluation of our proposal. We conclude and present our plans for future work in the final section.

2 Related Work

2.1 Association rule mining

The basis of frequent-pattern mining is association rule mining (ARM). Apriori (Agrawal and Srikant, 1994) is a prominent algorithm for ARM, basically consisting of two steps: the *join step,* where two elements of size *i-1* are merged, in order to form a new one, whose size is equal to *i*, provided that they have *only one item not in common*, and the *prune step*, in which candidate itemsets, having at least one infrequent subset of size *i-1*, are excluded from the frequent itemset of size *i*. Besides Apriori, many algorithms have been proposed, following its philosophy, ranging from *frequent-pattern mining* ones to *distributed and parallel extensions*, with some of them incorporating taxonomy information.

2.1.1 Frequent-pattern mining

A wide range of extensions of Apriori, aiming to tackle the extraction of the so-called *frequent patterns* or *episodes* has been proposed in the literature, varying from merely providing *sequential pattern mining*, where the order of occurrence of items matters, to requiring fulfilment of certain constraints, in order to take the involved patterns into account. In what follows, we are going to focus on two of them, namely *FP-Growth* (Han et al., 2004)*,* which does not generate candidate itemsets, but adopts a pattern-fragment growth method instead, as well as *GP-Close* (Jiang and Tan, 2006; Jiang et al., 2007), an algorithm, which adds *taxonomy constraints*, performing the so-called *generalized association rule mining* (GARM).

2.1.1.1 FP-Growth

The FP-Growth algorithm scans the transaction database twice: once to locate the frequent items of each user session[1], in order to store them in revised form (with its items in *descending global frequency* order), and another one, so as to generate a structure, which is called *frequent-pattern tree (FP-tree)* and is able to replace the database. In detail, for each revised transaction, a path in the FP-tree is created. However, in case the corresponding node for inclusion in the tree structure is already contained in an FP-tree path, its frequency is updated, without further altering the tree. Moreover, all appearances of the same item are connected with so-called *node-links*. Finally, an index called *header table* containing all frequent items sorted in descending global frequency order is created, which points to the first (i.e. leftmost) occurrence of each element in the tree. FP-Growth then uses this tree, in order to locate frequent patterns by scanning the header table from bottom to top, following the node-links. The items in the path from the root of the tree to the appropriate element form the *conditional pattern base* of the item and the minimum support of the items included in it determines the support of the suitable element. In fact, the conditional pattern base of each item can naturally be determined in a *bottom-up manner*, i.e. in *increasing total support order* [this is the approach followed by *FP-Growth*, as well as the *WAP* -web access pattern tree- algorithm in all circumstances; c.f. (Pei et al., 2000)].

The *CLOSET+* algorithm (Wang et al., 2003), which is also based on the frequent-pattern tree data structure, adopts a different mining technique: the aforementioned conditional pattern base construction manner is considered for mining dense datasets only (i.e. datasets with long patterns), while the one for items, which belong to sparse datasets (i.e. datasets, where each individual transaction contains only few of the available items), is created, using a *top-down pseudo tree-projection* method. The latter technique only is followed by the PLWAP algorithm (Ezeife et al., 2005). Furthermore, although the FP-tree is small enough to fit in main memory, there are cases where this is impossible, necessitating the use of *projection techniques* (either *parallel projection* or *partition projection*), in order for the appropriate conditional pattern base to fit in memory (Han et al., 2004), or the adoption of distributed schemes. In detail, the majority of distributed implementations of *FP-Growth* make the processors involved *exchange* some of their entries, prior to generating recommendations. For instance, in (Zaïane et al., 2001), a distributed implementation of *FP-Growth* in a shared-memory multiprocessor is considered, splitting the initial transaction database into (roughly) equal partitions, each one being assigned to a different processor, the latter following the two subsequent steps: *preparation for the construction of the FP-tree*, as well as *building of the FP-tree in each processor*. Moreover, a global header table is constructed, holding, for each element, its support in all machines, as well as the link to its leftmost occurrence in each FP-tree (Zaïane et al., 2001). As is probably expected, in order to generate frequent patterns, the FP-trees have to be mined, by creating the conditional pattern bases (Han et al., 2004) from the corresponding structure in each processor. A similar distributed extension of FP-Growth is the *PFP-tree* algorithm (Javed & Khokhar, 2004), with the exception of each FP-tree residing in the address space of the corresponding processor; the distributed

[1] A *user session* is considered to be all of the page accesses that occur during a single visit to a Web site (Cooley et al., 1999). According to Cooley et al. (1999) "the simplest method of achieving this (i.e. dividing page accesses of each user into individual sessions) is through a timeout, where if the time between page requests exceeds a certain limit, it is assumed that the user is starting a new session". Catledge and Pitkow (1995) have established a timeout of 25.5 minutes based on empirical data, while many commercial products use 30 minutes as a default timeout (Cooley et al. (1999)).

nature of PFP-tree requires the exchange of the conditional pattern bases of length *1*, so that each machine hosts an approximately equal database fragment.

Sucahyo et al. (2003) present an enhancement over the original FP-Growth algorithm using the *compact transaction tree* data structure, which avoids saving identical paths by storing their total number of appearances only in the first (leftmost) occurrence in the tree, thereby rendering further memory allocations for the same item combinations unnecessary. Since, after the creation of the parallel projections (Han et al., 2004) of the transaction itemsets, the created parallel projections can be partitioned among the processors and processed in parallel without requiring any extra communication step, each processor is able to generate the compact transaction tree, based on the parallel projection it receives and map it to a data structure called *Item-TransLink* (ITL), which is an even more compact format of storing the transaction database portion assigned (Sucahyo et al., 2003). The mining process from the compact transaction tree is straightforward and resembles the one of extracting frequent patterns from the FP-tree, based on the conditional pattern base concept (Han et al., 2004). After the mining process is complete, the master node (used initially for the distribution of the parallel projections) gathers the patterns generated at each processor and outputs the final frequent itemsets (Sucahyo et al., 2003). Finally, in case the projections are too large to fit into the memory of each processor, the parallel partition process is recursively applied, until the resulting fragments fit in the available processors.

El-Hajj and Zaïane (2006) tackle the parallelization of a variant of FP-Growth called *HFP-Leap* in a shared-nothing scenario, which requires two database scans. In the parallel implementation of HFP-Leap, during the first database scan each processor is assigned a set of transactions, with the different sets being approximately equal-sized; the local support counts are computed in each one, in order to be sent to the coordinating machine called *arbitrator* (it is not required for the arbitrator to be a special machine), which computes the global support values. In the second database scan, each machine calculates the support counts for each itemset assigned, and the support counts are sent to the arbitrator again, which accumulates the global values. Moreover, the local headerless trees for a subset of the elements are constructed in each processor, catering for load balancing. Furthermore, the final local maximals are communicated to the arbitrator, which prunes the itemsets having a frequent superset. Finally, the set of maximal patterns is outputted. To achieve load balancing in *HFP-Leap* (El-Hajj & Zaïane, 2006) considers three techniques namely *direct partitioning*, *round-robin partitioning* and *first-last*. In the *first-last* technique, each processor receives the most heavily loaded among the remaining elements, as well as the one, whose load is the least possible. Experiments have proved that the *first-last* scheme results in better load balancing, while in *direct partitioning* the machines assigned the first items will be the system bottleneck (Zaki et al., 1996).

Furthermore, (Yu et al., 2007) present *LFP-tree*, which is an improvement over *PFP-tree* (Javed & Khokhar, 2004), in terms of running time, load balancing, as well as speed-up ratio. In *LFP-tree*, a master node initially partitions the transaction database to the processors. The processors then calculate the (local) header table (local supports for the items) and send them to the master, in order for the master to generate the (global) header table (global item support). Each processor is currently ready to produce its FP-tree, according to the (global) f-list order (Wang et al., 2003) previously determined. Nevertheless, since there might be imbalances in the size of the trees created and, hence, in the mining time, a load degree evaluation function is used, based on the average depth of each item in the FP-trees, so as to lighten heavily-loaded processors, by means of exchanging FP-tree paths. After the exchanging process, FP-tree mining can proceed normally and the frequent patterns aggregated in the master processor (Yu et al., 2007).

Moreover, (Tanbeer et al., 2009) consider another PGDP distributed FPM approach, employing a novel tree structure, called *PP-tree* (Parallel Pattern tree) that significantly reduces the I/O cost by capturing the database contents with a single scan and facilitates the efficient FP-growth mining on it with reduced inter-processor communication overhead. In detail, the dataset is initially partitioned among the available processors, in order for each one of them to generate its (local) FP-tree, based on any (for instance, lexicographic) predefined order. Then, one processor is named as "master", collects the support counts for the items and stores them, according to the global frequency descending order [*global f-list order* (Wang et al., 2003)]. After the (global) f-list order gets known to the processors, they can rearrange their (local) FP-tree, according to the (global) f-list order employing an appropriate technique [e.g. branch sorting mechanism (BSM) which saves each unsorted branch into a temporary structure, where it is merge-sorted]. The time requirements are further reduced, thanks to the fact that not all FP-tree branches are unsorted (Tanbeer et al., 2009). This approach is more time-efficient than the alternative of a second database scan, after the global frequency descending order has been specified. After each FP-tree is rearranged according to the (global) f-list order, the processors are ready to perform frequent-pattern mining on their own tree structure. Nevertheless, since the real values of the support counts for the itemsets stored in other processors are not known, the observation that they cannot exceed the sum of the minimum values among the items comprising the itemset and the element(s), whose conditional pattern base is considered, is used, in order to generate the *potential global support*, a rather maximalistic approach on the actual itemset support value. This approach forces all itemsets, whose potential global support is less than the (global) support threshold to be pruned, which implies that there can be no false negatives (Tanbeer et al., 2009). After each processor has constructed its (local) frequent patterns, they are sent to the master node, in order for the actual support values to be computed. Thus, there are only three relatively small communication steps (sending the local item supports to the master node, the latter broadcasting the global f-list order, as well as the processors sending the local frequent patterns to the master node), which explains the superiority of *PP-tree*, in comparison to other parallel PGDP approaches, as far as the running time is concerned. The *PP-tree* approach is more suitable in dense, large datasets, low support thresholds and a substantial number of processors, compared to *LFP-tree* (Yu et al., 2007), with the gap augmenting, as the amount of processor increases, since *LFP-tree* requires a substantial time for interprocessor communication.

Another parallel FP-Growth implementation is depicted in (Li et al., 2008), where *FP-Growth* is split into five stages, namely (i) *sharding*, i.e. dividing the transaction database into successive parts, in order to store each one of them in a different computer; (ii) *parallel counting*, where the computation of the occurrence rate of the patterns in each shard is performed; (iii) *grouping items*, which takes place in a *single machine* and partitions the elements into groups called *G-lists*, in each one of which *FP-Growth* will be subsequently applied in the (iv) so-called *parallel FP-Growth* step, consisting of the *mapper*, as well as the *reducer* procedures. Moreover, we need to generate the final itemsets, considering the reducer information, which appears in all the shards; this is performed in the (v) *aggregating* step. In fact, the larger the dataset, the bigger the parallelizable portion of the algorithm; this implies that the so-called *PFP* (Parallel FP-Growth) algorithm is scalable. Nevertheless, *PFP* may result in *load imbalances*, as far as the G-lists stored in different computers are concerned; hence, the requirement that each processor's conditional pattern base satisfies a minimum path depth by continuing to generate the latter recursively, until the aforementioned condition is satisfied, is proposed in (Pramudiono & Kitsuregawa, 2003). However, the ideal minimum path depth remains an open issue.

Naturally, *sampling* the original transaction database is another workaround in parallel extensions of *FP-Growth*, trading accuracy for computational overhead (Pramudiono et al., 2005). Furthermore, the classification of the original database into a predetermined (usually 10) number of classes, so as to commence the construction of the conditional pattern bases in a special order is adopted in (Pramudiono et al., 2005), where it is reported that the ideal in terms of performance is the consideration of the third group (in decreasing frequency order) as the one to start generating the conditional pattern bases from. We simply note that, according to the aforementioned paper, sampling methods might perform better on data sets whose distribution is unknown.

Finally, another interesting distributed *FP-Growth* extension is depicted in (Dong et al., 2005), where the partitioning of the transaction database is based on the clustering obtained with the aid of a *neural network*. In detail, the so-called *CNFP* algorithm uses a 2-layered neural network, consisting of the input, as well as the competitive, layer, which are wholly connected to each other (i.e. each neuron of the input layer is joined with every neuron in the competitive layer), in order to determine the corresponding FP-subtree the appropriate transaction must be inserted into. The *CNFP* algorithm consists of the following 5 steps: *initialization*, *competition*, *FP-subtree construction*, *CNFP-Growth*, as well as *global frequent-pattern construction*.

2.1.1.2 Adding taxonomy constraints: the GP-Close algorithm

Taking the taxonomic characteristics of web 2.0 applications into account, it is worth considering the hierarchical structure of the categories, to which items belong. A well-known algorithm, which in fact performs *generalized association rule mining* (GARM),[2] is *GP-Close* (Jiang and Tan, 2006; Jiang et al., 2007). Contrary to some initial proposals in the field [e.g. Basic (Srikant and Agrawal, 1995)], GP-Close eliminates *over-generalized* (i.e. redundant) *patterns* by applying two efficient pruning strategies, *child-closure pruning*[3] and *subtree pruning*, the former getting rid of redundant closures, *wherever they are encountered* in the tree structure that is constructed[4] and the latter taking care of the removal of subtrees, whose root corresponds to an over-generalized node. It should be noted that the elimination of redundant patterns is proven in (Jiang and Tan, 2006; Jiang et al., 2007) to be equivalent to *closed pattern mining*, that is, looking for itemsets, which have *no proper superset, whose support value is identical* to the one of the itemset itself.

Considering the power of the FP-Growth algorithm previously presented, it is really beneficial to combine it with a GARM approach: (Pramudiono and Kitsuregawa, 2004) suggest extending the information stored in the appropriate frequent item list with each element's ancestors in the taxonomy structure, while Giannikopoulos et al. (2010) propose the incorporation of the frequent-pattern tree instead of the original transaction database into GP-Close, which is able to provide a solution to the *concept drift* issue (Tsymbal, 2004). This corresponds to the fact that users' interests change along the time axis, rendering the naïve approach of recommending the frequent itemsets only inadequate, since the newly-arrived documents, belonging to an important category, would fail to be proposed, despite being the most interesting for the average user; this is particularly true in news-related sites, where new content is constantly added. Moreover, as far as the creation of the hierarchical structure to be used as input to the GARM algorithm is concerned, we could either use one explicitly provided (e.g. portals typically use a categorization for the articles they publish), or extract it from the documents involved, using a similarity metric as indication of the point the newly-considered concept must be inserted into [the new term can be placed as a child of either (a) a node whose similarity distance is less than a threshold or (b) the upper-level (pseudo)root,

2 GARM was first presented in (Srikant and Agrawal, 1995).

3 Child-closure pruning halves the search space, provided that it is applied as early as possible (i.e. at the children of the root).

4 This tree structure is known as *closure enumeration tree* (Jiang and Tan, 2006; Jiang et al., 2007).

holding all elements together, if no such node is identified] (Heymann & Garcia-Molina, 2006). In fact, Heymann & Garcia-Molina (2006) utilize the tagging information of the documents, constructing a hierarchy of tags. An alternative approach, proposed in (Dakka & Ipeirotis, 2008), is to extract common terms from the texts (these expressions form the original database), in order to enhance them with external synonyms and/ or more general concepts (hypernyms), using an appropriate resource (e.g. WordNet, Wikipedia); the result of this process is the construction of the *contextualized database*.[5] The terms which appear frequently in the latter but *not* in the original database, are considered *facets* (that is, words or collocations that characterize the document) and are transformed to hierarchies [using the subsumption algorithm presented in (Sanderson & Croft, 1999), for instance], thereby producing a taxonomy structure. Finally, an overview of distributed generalized association rule mining algorithms (prior to *GP-Close*) is presented in (Shintani & Kitsuregawa, 1998), where six algorithms (*NPGM*, *HPGM*, *H-HPGM*, *H-HPGM-TGD*, *H-HPGM-PGD*, *H-HPGM-FGD*) are outlined, whose main difference lies in the method the initial transaction database (kept in *one* processor as a whole) is partitioned among the available computers.

2.2 Peer-to-peer systems

A wide range of peer-to-peer (P2P) systems has been proposed in the literature, varying from *unstructured* approaches (Kirk, 2003), which exhibit *no specific topology,* as far as the participating peers are concerned, to *structured* ones such as Chord (Stoica et al., 2001), CAN (Ratnasamy et al., 2001) and P-Grid (Abererer al., 2005), which take extra care of the preservation of the network topology during updates, that guarantees correct answers later on, without excessive bandwidth consumption; hence, the majority of contemporary P2P systems, distributed hash tables (DHTs) included, fall into this category. Furthermore, *hierarchical* applications such as Edutella (Nejdl et al., 2002) and P2P-DIET (Idreos et al., 2004), as their name implies, partition their nodes into two disjoint sets: *clients*, which publish documents or submit queries, as well as *super-peers*,[6] which store information about the files, located in the underlying clients. Furthermore, P2P systems can be further categorized into *centralized* -e.g. Napster (Napster website, 2011)-, which distinguish a peer, that stores an index of the rest ones, as well as the resources the latter are willing to share and *decentralized* [Gnutella (Kirk, 2003)],[7] where there is no need for a master node.

3 The Distributed Recommender System

Our implementation corresponds to a distributed recommendation system, consisting of *peers*, each one storing a portion of the end users' clickstream log files. Peers are information providers, and each one's log file corresponds to requests for information items that end users have made from them. A peer may specialize in a certain taxonomy topic, in which case log entries within each peer include items from the specific taxonomy topic only. This peer setup is common in digital libraries [e.g. (Maniatis et al., 2005; Ahlborn et al., 2002)], social media networking [e.g. (Buchegger et al., 2009; Pouwelse et al., 2007; Tsai et al., 2007)], news agencies [e.g. (Jelasity and van Steen,

5 An analogous method is described in section 3 of (Jiang et al., 2007).

6 Super-peers can be further split into hierarchies in turn, as far in depth as we wish; however, it is rare to trace an application, whose super-peers are not all equal among themselves.

7 The two major classifications of P2P networks presented above are *not* mutually exclusive; that is, an implementation might belong to an alternative from each one simultaneously (e.g. Gnutella is the prototypical *unstructured* and *decentralized* system).

2002; Voulgaris et al., 2005; Deuze et al., 2007)], server farms [e.g. (Wang and Dasgupta, 2005; Li et al., 2008b)], content distribution networks [an extensive survey can be found in (Androutselis-Theotokis and Spinelis, 2004)] etc. In this context, opting for a peer-to-peer architecture for implementing the recommender system over the underlying content provisioning network is a natural choice, since it allows for exploitation of already available network topology knowledge and reuse of network connections, while the participating nodes have also established trust relationships of some level between them, which may be extended to cover the data exchanged for the purposes of the recommender system.

Within such a peer network, a major issue affecting the operation of a recommendation algorithm is the fact that information regarding any particular session (cf. footnote 1 in subsection 2.1.1.1) may be scattered in the log files of multiple peers; therefore, performing frequent-pattern mining in individual log files will miss patterns that *do* belong to the same session but whose clickstream traces are in different log files. For instance, in the setup illustrated in fig. 1, it is possible that a user has requested some pages related to sports (with the relevant clickstream information being stored in the log file of peer 1) and some pages related to international politics (with the relevant clickstream information being stored in the log file of peer 3); if log files are mined in isolation, it is impossible to detect any pattern containing *both* sports-related *and* international politics-related pages. Therefore, *session reconstruction* is a first issue that has to be tackled. While transferring the full log files to a single server may deal with the session information scattering issue straightforwardly, this approach is effectively a centralized one, exhibiting the performance and scalability problems presented above. Hence, we have opted for a distributed solution to the problem, which comprises of the *subperiod assignment phase,* the *log exchange phase*, the *local mining phase* and the *pattern/rule exchange phase*, which are executed at each peer sequentially. These phases are described in the following subsections.

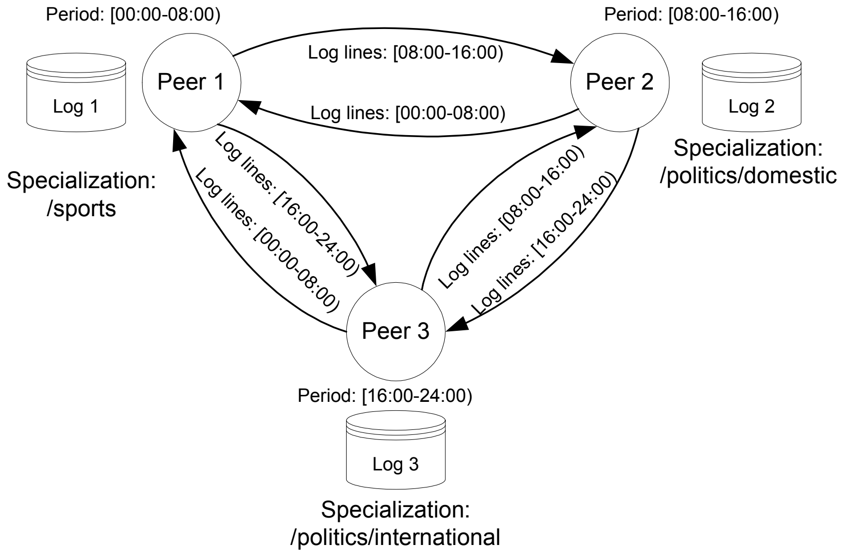

Fig. 1. Peer specialization and log exchange between peers

3.1 The subperiod assignment phase

The first phase of the distributed algorithm is the *subperiod assignment phase*, within which the time period covered by the logs is split into *N* (approximately) equal sub-periods (*N* is the number of peers within the P2P system), and each peer is assigned one such sub-period for subsequent processing. Partitioning is always performed preserving half-hour boundaries, e.g., in order to split a 24-hour period among 5 peers, the assignment {P1: [00:00-05:00), P2: [05:00-10:00), P3: [10:00-15:00), P4: [15:00-19:30), P5: [19:30-24:00)} is used, instead of the assignment {P1: [00:00-04:48), P2: [04:48-9:36), P3: [9:36-14:24), P4: [14:24-19:12), P5: [19:12-24:00)} to minimize the cases that some sessions are split among different peers. Note that splitting sessions among peers is not entirely avoided, since, for instance, in the example above, a session beginning at 19:05 and ending at 19:34 will be split among peers P4 and P5, under both the adopted and the alternative partitioning scheme. This may lead to missing some frequent patterns and consequently some association rules. We have evaluated this side-effect of the partitioning scheme and we have found it to be limited to tolerable levels (< 12%); details of this evaluation are given in section 4.

In the current implementation, assignment of time periods to peers is coordinated by a super-peer called *arbitrator*, where all peers register; the former splits the overall time period to the appropriate time segments and then publishes the assignments to all peers. This setup has been adopted for implementation convenience, and it is straightforward to assign the time period partitioning responsibility to the peer network (i.e. without the need for a super-peer), where a coordinator for the partitioning activity can be co-decided using any distributed election algorithm (Tanenbaum and van Steen, 2006).

3.2 The log exchange phase

Once sub-period assignment has taken place, the *log exchange* phase commences, where each log file entry is transferred to the peer that has been assigned the corresponding time interval. At the end of the *log exchange* phase, each peer has obtained all log records that pertain to the period it has been assigned.

Fig. 1 shows an example where three peers specializing in various topics of the taxonomy exchange their log records. Note that topic specialization may not be limited to the same level of the taxonomy (in the example, one peer specializes in topics of a top-level branch, whereas the other two in topics of branches at level 2); moreover, a peer may include topics from the whole taxonomy (i.e. not specialize in any particular topic, such as a generic news agency). At the first step, the period covered by the logs (assumed to be 24 hours) is split into 3 equal-sized subperiods, and afterwards the log records are exchanged as illustrated by the labelled arrows.

While the log exchange phase is executed at each peer, a separate task receives lines sent by other peers and appends them to the local *FGP-input* file. At the end of this stage, the *FGP-input* file at each peer contains all web log lines pertaining to the timeline segment assigned to the particular peer.

Since the volume of the information exchanged between peers in this phase may be large, introducing therefore significant communication overhead, the following techniques may be employed to minimize the volume of data transferred between peers:

1. A log entry typically includes the item request timestamp, the requesting IP and the item identifier. Items within a single session can be grouped by requesting IP, and therefore the address is transmitted only once. For frequent-pattern mining algorithms, where the *order of item appearance* is *not* significant, it is possible to exchange session batches, with each batch being labelled with the requesting IP and an identification of the particular half-hour, and using the item identifiers as the

batch contents, avoiding the transmission of individual timestamps as well. Note that in many content provisioning systems (e.g. content management systems and content distribution networks), items are characterized by an id, and it suffices to include this id in the exchange phase, instead of the whole request URI. Furthermore, the sizes of the requesting IP, the timestamp and the item identifier are comparable; therefore, these grouping techniques may reduce the size of the exchanged information by approximately 33% (if only grouping by IP is employed) or 66% (if *both* grouping by IP *and* timestamp substitution by half-hour designation are used).

2. If the number of accesses for different periods is not uniform among the peers {e.g. peer 1 has considerably more accesses during the time period [08:00-16:00)}, it would be beneficial for each peer to be assigned the time period for which it has a more extensive log file. This can be accomplished by having the peers exchange *histograms* (Poosala, 1997), where the overall number of log entries for each half-hour period is recorded; the histograms will then be used for computing a partitioning that minimizes data transfer. This goal can be better served by using a more flexible time period partitioning technique, allowing for non-consecutive periods to be assigned to peers. We could even make each peer store information, with regard to certain half-hour periods, instead of insisting on time intervals {i.e. peer 1 could store entries belonging to the first [00:00-00:30), the fourth [01:30-02:00), as well as the seventh time period [03:00-03:30)}. Any method for achieving histogram-based load balancing in peer-to-peer systems -e.g. (Vu et al., 2009; Scholl et al., 2007)- can be used.
3. Finally, to reduce the time requirements of the log exchange step, the web log entries could be *compressed*, prior to sending them to the corresponding peer.

The last two enhancements to the basic log exchange scheme have been applied in our implementation and result in minimization of the data transfer among peers; hence, the algorithm's overall execution time is considerably reduced, as indicated by the performance metrics presented in section 4.

3.3 The local mining phase

The log exchange phase was performed, in order to make each peer store the web log entries belonging to the time intervals it specializes in; hence, each node is subsequently ready to apply a frequent-pattern mining algorithm. In fact, given the inherent taxonomic nature of social data, it would be better to utilize the FGP algorithm presented in (Giannikopoulos et al., 2008; Giannikopoulos et al., 2010) which enhances the performance gains of FP-Growth (Han et al, 2004) with information about the categories the articles belong to, by incorporating GP-Close (Jiang and Tan, 2006; Jiang et al., 2007). The frequent patterns produced by each peer during the frequent-pattern mining phase are tagged with the number of appearances of each one; however, this number counts only the appearances of the corresponding pattern in the time segment(s) that the peer has been assigned, and, therefore, the need for summing up all the counts produced by different peers arises. This is accomplished in a distributed fashion, within the pattern/rule exchange phase.

3.4 The pattern/rule exchange phase

Within the local mining phase, the FGP algorithm at each peer produces the frequent itemsets mined from its local *FGP-input*, with each pattern being tagged with the number of its local appearances (its *support* (Giannikopoulos et al., 2008)). It is possible however that the same pattern has been produced by multiple peers, and therefore each peer has only a *partial value* of the pattern's support. Since the support metric, by virtue of its definition, effectively counts each itemset's number of appearances, it directly follows that the *global value* of the support of some pattern *p* is the sum of all *partial support values* that have been produced by all peers for the same pattern.

A straightforward method for computing the sum of all *partial support values* for the generated itemsets is to collect to a single peer all frequent itemsets produced by all peers, and then have this peer calculate the sums of partial support values for each distinct itemset. This solution however is effectively centralized, exhibiting scalability problems and introducing a single point of failure.

In order to accomplish the calculation of the itemsets' global support value in a distributed fashion, each peer iterates over the itemsets it has produced, computing a hash value over the elements of the itemset (the range of the hash function is [*1..N*] where *N* is the number of peers in the network), and forwarding all frequent itemset records with hash value equal to *i* to the *i-th* peer; then the appropriate peer simply sums up the number of appearances of all records having a common itemset.

Since the frequent-pattern mining phase is performed for producing association rules to be exploited later for formulating recommendations, a frequent pattern of the form $\{i_1, i_2, \ldots, i_n\}$ can be viewed as a rule $[(i_1, i_2, \ldots, i_{n-1}) \rightarrow i_n]$ if item order is significant (i.e. if a user has viewed items $i_1, i_2, \ldots, i_{n-1}$ then item i_n is recommended), or as a set of rules $\{[(i_1, i_2, \ldots, i_{n-1}) \rightarrow i_n], [(i_1, i_3, \ldots, i_n) \rightarrow i_2], [(i_2, \ldots, i_n) \rightarrow i_1]\}$ if the order of items is not significant. For each such rule, we are able to compute a hash value on its antecedent, which can be used to determine the peer where it will be forwarded. Each peer computes the hash value of all the rules it has computed, and then groups rules by their destination, each such group being sent to the respective peer in a single message. Subsequently, when a recommendation needs to be formulated for a user that has viewed items $\{i_1, i_2, \ldots, i_{n-1}\}$, the hash value for this set can be computed and the peer corresponding to it can be queried to produce the required recommendation.

3.5 Limiting recommendations within topics

The scheme presented above for log file distribution and frequent-pattern mining allows for formulating association rules that span across topics of the taxonomy – e.g. if a user has viewed a set of pages in the *politics* category, then a page from the *sports* taxonomy branch may be recommended. While this could be desirable for some cases (e.g. a generic portal), in some other cases –such as a specialized blog aggregator– we might intend to limit the recommendations to certain branches of the taxonomy. Under such circumstances, it suffices to restrict the sources during the log exchange phase to the peers specializing in the specific branches of the taxonomy, while it is not necessary to limit the peers participating in the frequent-pattern mining phase to this subset only. For instance, in the example of fig. 1, the logs of peers 2 and 3 (subtopics of the *politics* branch) can be distributed among all peers and processed, so as to produce association rules, whose both the antecedent and the consequent are limited to the *politics* branch. This property stems from the hierarchical nature of our scheme, according to which each peer stores links to posts belonging to a specific topic, and is particularly beneficial when *subscription interests* of the users are available, and, therefore, limitation of recommendations to specific topics is implied.

The pseudocode of distributed FGP is outlined in the following figures:

```
determine time period(s) for which the peer is responsible, in
     cooperation with other peers
do in parallel
     preprocess own file, storing in local file clickstreams the
           peer is responsible for and distributing other
           clickstreams to respectively responsible peers
     receive clickstream information sent from other peers and
           append them to local file
end parallel /* wait for both parallel tasks to finish; now
     local clickstream file contains all clickstream entries for
     period(s) of peer responsibility */
runFGP /* on local clickstream file */
generate and exchange rules
```

Fig. 2. The pseudocode each peer running distributed FGP executes.

The pseudocode of task *preprocess own file*, which realizes the *log exchange* step, as well as *generate and exchange rules*, which sends all appearances of a rule (no matter the peer, where it has been generated) into only one node, hashing on its antecedent, so that the occurrence metrics are accumulated, are presented in fig. 3 and 4, respectively:

```
preprocess own file {
    read web log
    foreach(l log line){
        if(myOwn(l)){
            write to my log
        }
        else{
            find specializing peer P and send l to P
        }
    }
}
```

Fig. 3. Pseudocode for the *preprocess own file* task, which caters for each peer receiving the log entries belonging to its specialization period(s).

where function myOwn determines if the corresponding web log line *l*, which is passed as its sole input argument, must be processed by this peer.

4. Experimental Evaluation

We have implemented the distributed version of FGP in Java 1.6 and evaluated it on a P2P network, consisting of a varying number of peers, ranging from 2 to 24, to (a) compare its execution time with the one of the centralized implementation (Giannikopoulos et al., 2008; Giannikopoulos et al., 2010), as well as to evaluate its scalability and (b) assess the quality of recommendations offered by the algorithm since, as stated in section 3.1, due to session splitting among peers, some frequent patterns and association rules may be missed. The CPU frequency of the peer nodes is 2.6 Ghz, while the available memory at each node equals 2GB. We have first of all experimented with two log files of varying sizes, in order to quantify the performance gains obtained for different-sized log files. Each experiment was run thirty times, and then an average over the thirty runs was computed. The nodes are interconnected through an 100Mbit switched Ethernet medium at the basic setup, a 10Mbit shared Ethernet setup being furthermore used in a second set of experiments to assess the effect of the communication medium speed on the algorithm performance.

```
generate and exchange rules {
    generate rules from FGP's frequent itemsets
    foreach (peer P)
        rules(P) = ∅
    foreach(rule r in generated rules){
        destination =hashCode(r.antecedent) % NumPeers;
        rules(destination) = rules(destination) ∪ {r};
    }
    foreach (peer P ≠ self) {
        send rules(P) to P
        receive incomingRules(P) from P
        foreach (rule r′ in incomingRules(P)){
            if (rules(self) contains r where (r′.antecedent =
r.antecedent ∧ r′.consequent = r.consequent))
                r.support = r.support + r′.support;
            else
                rules(self) = rules(self) ∪ {r′};
        }
    }
}
```

Fig. 4. The rule generation and exchange task.

4.1 Performance evaluation

Table 1 illustrates the experimental results obtained from executing the distributed algorithm on a medium-sized log file without applying the histogram-based load balancing presented in section 3.2, using 2-24 peers. Respectively, Table 2 depicts the results from executing the distributed algorithm on the same log file, using load balancing to distribute web log partitions more evenly among the peers. In each table, column "#lines initial" accounts for the number of lines that each peer holds prior to executing the distributed algorithm, while column "#lines after exchanging" corresponds to the number of lines that are assigned for processing to the peers after the log exchange step. The time needed for the aforementioned phase is depicted in column "data exchange time", whereas the one required by the peers to perform the pattern mining phase is illustrated in column "CPU time". Finally, column "total time" accounts for the algorithm completion time, and column "%gain" shows the speedup against the centralized version of the algorithm. For columns "#lines initial", "#lines after exchanging", "data exchange time" and "CPU time", we list the average, minimum and maximum measurements for all peers in the system, to show the effect of load balancing.

As can be seen from the figures, the algorithm scales well with the number of peers, with the algorithm execution time dropping approximately 40% (with a maximum theoretical decline of 50%) as the number of peers doubles [note that in the last two configurations (16 and 24 peers), the number of peers increases by 50% only, not 100%; therefore, in this case, the theoretical drop is 33% and the observed reduce rate in the execution time reaches 24.34% on average]. The communication overhead for the data exchange phase is very low, accounting for no more than 8.16% of the algorithm execution time and, as can be seen from the table, the communication cost remains fairly stable as the number of peers increases. This can be explained by considering that, on increasing the number of peers, each of them starts off with a smaller portion of the overall log size (*log_size/N*, assuming a perfectly uniform initial distribution). Then each peer sends a ratio *(N-1)/N* of this log size to other peers (again supposing a perfectly uniform distribution regarding the log lines exchanged with other peers); therefore, the data sent by each peer is equal to $[log\_size*(N-1)/N^2]$, which decreases with the value of *N*. On the other hand, the peer network setup time increases slightly with the number of peers (more connections need to be established), and, therefore, the sum of these differences leads to having very small variations on the overall communication time. Finally, the load-balanced version achieves 13.04%-28.09% better results, compared to its non load-balanced counterpart, as is depicted in Table 2.

#peers		#lines initial	#lines after exchanging	Data exchange time (sec)	CPU time (sec)	Total time (sec)	%gain
1		345,610	345,610	-	601.45	604.32	-
2	avg	172,805	172,805	1.66	300.76	430.34	40.43%
	min	172,106	120,661	1.57	107.72		
	max	173,504	224,949	1.70	428.64		
4	avg	86,402	86,402	1.38	83.78	172.87	249.58%
	min	86,079	38,205	1.30	17.94		
	max	86,673	130,197	1.66	171.25		
8	avg	43,201	43,201	1.49	36.54	102.02	492.35%
	min	42,626	12,606	1.42	4.15		
	max	43,566	76,986	1.62	100.56		
16	avg	21,600	21,600	1.77	21.12	58.01	941.75%
	min	21,328	5,107	1.23	2.64		
	max	21,931	40,550	2.21	56.74		
24	avg	14,400	14,400	1.43	17.08	46.93	1,187.71%
	min	14,210	3,251	1.32	2.89		
	max	14,590	27,701	1.53	45.56		

Table 1. Experimental results for the medium-sized log file without load balancing

Table 3 and Table 4 depict the same measurements obtained from executing the distributed algorithm on a large-sized log file, with the number of peers also varying from 2 to 24. The non-balanced version exhibits the same behaviour as the medium-sized log file regarding its performance, i.e. there is an approximate 40% drop-off on doubling the number of peers, as far as the algorithm execution time is concerned. In the load-balanced version, the performance gains are higher, with the execution time dropping approximately 58% when doubling the number of peers. This can be explained by taking into account that, as indicated by the benchmark results presented in (Said et al., 2009), FP-Growth exposes super-linear sizeup ratio (that is, its execution time increases significantly, as the size of the log to be processed augments) and, therefore, achieving a better partitioning (through load balancing) for a big log file results in considerable performance gains, while the benefits stemming from doing so for a smaller log file are more limited. Therefore, the load-balanced version in this case achieves a performance improvement ranging from 9.92% to 48.02%, compared to its non load-balanced counterpart. Regarding the communication cost, the remarks made for the medium-sized log file hold in this case too; nevertheless, the upper bound of the quotient of the *communication_cost* to *total_time* is considerably smaller (approximately equal to 1.9%, while the upper bound of the corresponding ratio for the medium-sized web log is roughly 3.5%).

#peers		#lines initial	#lines after exchanging	Data exchange time (sec)	CPU time (sec)	Total time (sec)	%gain
1		345,610	345,610	-	601.45	604.32	-
2	avg	172,805	172,805	1.62	257.30	374.21	61.49%
	min	172,106	166,989	1.56	120.44		
	max	173,504	178,621	1.67	372.65		
4	avg	86,402	86,402	1.37	72.90	124.31	386.14%
	min	86,079	79,348	1.32	48.58		
	max	86,673	92,439	1.43	122.99		
8	avg	43,201	43,201	1.48	35.48	79.16	663.42%
	min	42,626	19,676	1.31	12.67		
	max	43,566	50,213	1.59	77.66		
16	avg	21,600	21,600	1.81	23.77	46.94	1,187.43%
	min	21,328	13,487	1.18	10.49		
	max	21,931	27,560	2.03	45.50		
24	avg	14,400	14,400	1.24	16.43	40.30	1,399.55%
	min	14,210	5,509	0.99	4.24		
	max	14,590	22,838	1.40	39.07		

Table 2. Experimental results for the medium-sized log file with load balancing

#peers		#lines initial	#lines after exchanging	Data exchange time (sec)	CPU time (sec)	Total time (sec)	%gain
1		1,213,770	1,213,770	-	2,229.47	2,232.75	-
2	avg	606,885	606,885	3.84	1,033.13	1,609.53	38.72%
	min	606,689	408,430	3.77	459.56		
	max	607,083	805,340	4.01	1,605.71		
4	avg	303,442	303,442	2.73	435.69	888.78	151.22%
	min	302,345	120,685	2.70	72.36		
	max	304,144	468,338	2.76	886.06		
8	avg	151,721	151,721	2.54	180.99	496.96	349.28%
	min	151,076	36,518	2.70	72.36		
	max	152,232	277,127	2.76	494.21		
16	avg	75,860	75,860	2.05	79.53	263.62	746.96%
	min	75,378	16,234	1.90	3.38		
	max	76,144	148,456	2.18	261.64		
24	avg	50,573	50,573	1.73	55.81	191.99	1,062.95%
	min	50,200	10,220	1.68	4.50		
	max	51,002	97,786	1.90	190.29		

Table 3. Experimental results for the large-sized log file without load balancing

#peers		#lines initial	#lines after exchanging	Data exchange time (sec)	CPU time (sec)	Total time (sec)	%gain
1		1,213,770	1,213,770	-	2,229.47	2,232.75	-
2	avg	606,885	606,885	3.59	1,168.75	1,449.86	54.00%
	min	606,689	595,182	3.57	891.441		
	max	607,083	618,588	3.64	1,446.26		
4	avg	303,442	303,442	2.69	499.77	756.01	195.33%
	min	302,345	252,304	2.60	230.56		
	max	304,144	328,525	2.78	753.38		
8	avg	151,721	151,721	2.64	191.36	323.34	590.53%
	min	151,076	74,685	2.45	38.39		
	max	152,232	177,959	2.79	320.87		
16	avg	75,860	75,860	1.96	74.51	137.04	1,529.27%
	min	75,378	32,685	1.76	19.76		
	max	76,144	104,838	2.28	134.96		
24	avg	50,573	50,573	1.84	50.56	114.22	1,854.78%
	min	50,200	5,048	1.63	4.36		
	max	51,002	77,850	2.11	112.23		

Table 4. Experimental results for the large-sized log file with load balancing

In Figure 5, the algorithm execution time against the number of peers for the two different log file sizes and the two log distribution policies (non-balanced and balanced) is graphically illustrated.

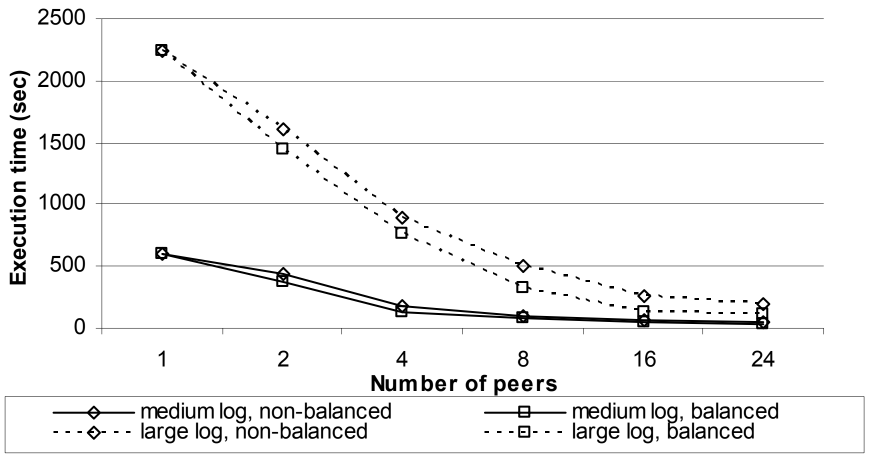

Fig. 5. Algorithm execution time against number of peers

To further investigate the effect of the communication infrastructure on the algorithm performance, the same experiments were run a second time substituting the 100Mbit switched Ethernet infrastructure with a 10Mbit shared Ethernet hub, and the results of this experiment are depicted in Figure 6. For graph clarity purposes, only the data regarding the maximum communication times of the load-balanced version of the algorithm is illustrated and compared with the data obtained from the experiments run on the 100Mbit switched Ethernet infrastructure. As can be seen in the graph, although the theoretical network capacity decreases 10 times, the corresponding increment in communication time varies between 2.5 and 3.8 times, approximately. This indicates that in the 100Mbit scenario, the communication means was not exploited to its full capacity, mainly due to the compression applied to the log data prior to sending it over the network. The results from the 10Mbit scenario verify that the proposed algorithm is applicable in wide area networks too, since nowadays 10Mbit lines are a commodity for connecting servers to the internet.

4.2 Quality of recommendations assessment

In order to assess the quality of recommendations offered by the proposed algorithm, we quantify the *rule matching percentage* against the centralized version of the algorithm (which produces all association rules that should be generated). As noted above, since the partitioning scheme may lead clickstream information pertaining to the same session to be processed by different peers, some frequent patterns may be missed, and this will in turn result in failing to produce the related association rules. Formally, for an experiment producing $\#rules_{exp}$ rules, the rule matching percentage is defined as $\#rules_{exp} / \#rules_{centr}$, where $\#rules_{centr}$ is the number of rules produced by the centralized version of the algorithm. Note that, since the frequent patterns identified by the distributed versions of the algorithm are a *subset* of those identified by the centralized algorithm (since patterns can be only *missed*), it is not possible for the distributed algorithms to generate patterns that are not produced by the centralized one. As the number of peers increases, more sessions are bound to be split among different peers and therefore the rule matching percentage is expected to drop.

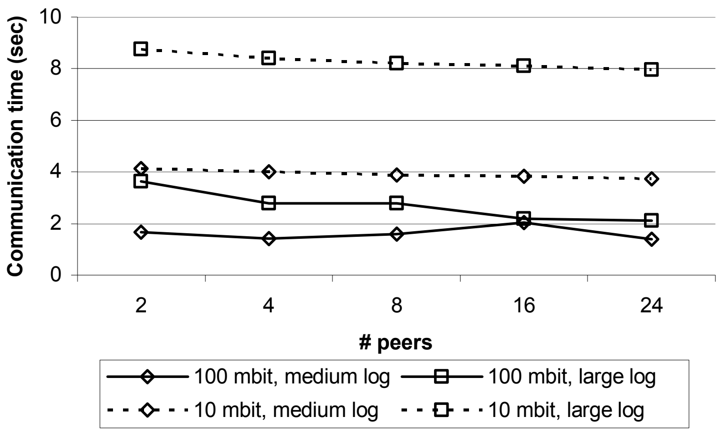

Fig. 6. Communication time for experiments running on varying network infrastructure

Fig. 7 illustrates the rule matching percentage for different numbers of peers (2, 4, 8, 16 and 24) and considering both the unbalanced and the load-balanced version of the proposed algorithm. As can be seen from the diagram, the matching percentage is found to be at quite satisfactory levels (> 88%), with the minimum value being observed for the load-balanced version of the algorithm when the number of peers is set to 24. The load-balanced version of the proposed algorithm always exhibits lower matching percentage than its non load-balanced counterpart; this is to be expected, since the non load-balanced version always assigns consecutive half-hour periods to each peer, therefore creating fewer points where session splitting may occur.

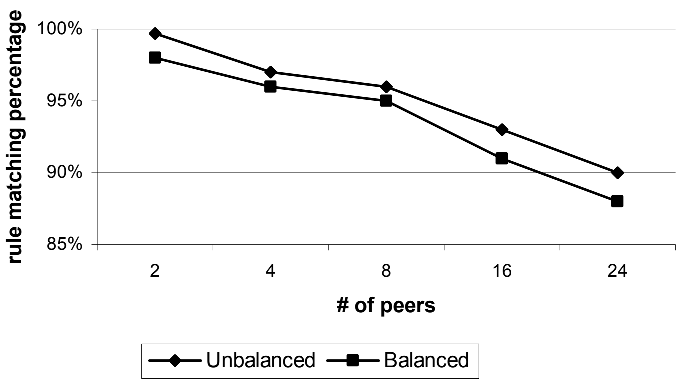

Fig. 7. Matching percentage of recommendation rules

Finally, we note that the experiments were run using 30 different random partitionings of the initial log file among the peers, and in all cases the matching percentage was in close agreement to the one illustrated in Fig. 7 ($|x - \overline{x}| < 0.7\%$, where x is any individual measurement and $\overline{x}$ is the measurement mean, shown in Fig. 7).

4.3 Comparison with other distributed mining approaches

First of all, it must be stressed that the particular mining technique followed in *distributed FGP* combines many contemplations and, therefore, to the best of the authors' knowledge, there is no other implementation in the literature, taking all these characteristics into account. Our technique is in essence a *distributed*, *taxonomy-aware pattern growth and database projection* (PGDP) algorithm, which works in a *shared-nothing* architecture. The applications, which are closer to our approach, are *parallel* (shared-memory), as well as *distributed FP-Growth* implementations outlined in section 2.1.1.1, whose time requirements will be compared to our work.

The programming easiness of the *shared memory* scheme, compared to the *cluster* environment, explains the early appearance of *parallel* FP-Growth extensions. One of the most prominent ones is *MLFPT* (Zaïane et al., 2001), which obtains a speedup up to *15.04*, in a *16*-processor scenario and a comparable dataset (1M transactions, roughly equal to the amount of transactions of our big university web log file), which is slightly worse than our approach (*15.29*), not to mention the better frequent patterns generated, due to the application of taxonomy data in the mining process in *distributed FGP*.

The *par-ITL* algorithm presented in (Sucahyo et al., 2003) has a significant speedup, as the number of processors increases, but only up to *12* machines. This approach is proved to be scalable in dense datasets, containing many transactions. Furthermore, El-Hajj & Zaïane (2006) report a speedup gain of *8*, while running on *16* processors;

nevertheless, their *Parallel Leap* algorithm performs maximal-pattern mining (i.e. *not* all frequent patterns are outputted), while it does not make use of any taxonomy information.

Furthermore, the *LFP-tree* approach presented in (Yu et al., 2007) achieves a speedup ratio of *4.5* in a dataset similar to our small university web log file. The equivalent figure for *distributed FGP* is *11.87.* Moreover, the state-of-the-art in PGDP-based distributed FPM algorithms is presented in (Tanbeer et al., 2009), which uses the *PP-tree* data structure and achieves a speedup of *4* for *4* processors in a dataset of *100000* transactions, while our implementation achieves a speedup ratio equal to *6.63* when running in our small university web log file, which contains an analogous number of entries.

In short, *distributed FGP* outperforms other approaches proposed, while running on a shared-nothing scenario and using the taxonomy information inherent in FPM, in order to improve the frequent patterns generated.

5. Conclusions

In this paper we have presented an extension of the FGP algorithm into the P2P domain, which allows distributed web logs to be mined efficiently for frequent patterns in a shared-nothing scheme. A further extension was the incorporation of log distribution according to histogram statistics to achieve better load balancing and smaller algorithm completion times. The proposed extensions have been implemented and benchmarked, using log files of various sizes and different numbers of peers, and the performance gains have been quantified in different network settings; we have also assessed the quality of recommendations produced by the proposed algorithm. It would be interesting to enhance our approach, in order to take into account the information regarding the taxonomy and node specialization, since in the current version of the algorithm, items belonging to different branches of the taxonomy are not differentiated. Hence, it is possible that when a user navigates within the *politics* branch, for instance, a recommendation to a page of the *sports* category is produced, if past users had visited pages from both categories. Typically, however, recommendations are offered only within the categories that a user has visited; this can be achieved by extending node specialization in the GARM phase, so as to have nodes specializing in *both* topic categories *and* time intervals. Besides limiting the recommendations only within the categories that a user has visited, this approach allows the system to scale in the GARM phase beyond its current limit of 48 peers (one peer per half-hour period). This is particularly useful in situations where the number of available content providers exceeds 48, e.g. content distribution networks. The incorporation of topic categories into the distribution scheme may also lead to further reducing the need for data exchange, since each peer can be set to process mostly the data it specializes in. Finally, we will investigate the incorporation of privacy-preserving data mining techniques into the distributed data mining architecture. Currently a number of methods are proposed in the literature, e.g. (G. Qiong and C. Xiao-hui, 2009; M. Kantarcioglu and C. Clifton, 2004; C. Su and K. Sakurai, 2008); nevertheless, these approaches work either in different data partitioning schemes or over different architectures. We will therefore investigate the adaptation of these algorithms to fit the data mining scheme proposed in this paper.

References

Aberer, K., Datta, A., Hauswirth, M. and Schmidt, R. (2005). Indexing Data-oriented Overlay Networks. *Proceedings of the 31st international conference on very large databases*, Trondheim, Norway.

Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th international conference on very large databases*, San Francisco, California, 487-499.

Ahlborn, B., Nejdl, W. and Siberski, W. (2002). OAI-P2P: a peer-to-peer network for open archives. *Proceedings of the international conference on parallel processing workshops,* 462 -468.

Androutselis-Theotokis, S. and Spinellis, D. (2004). A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371.

Buchegger, S., Schiöberg, D., Vu L.H. and Datta, A. (2009). PeerSoN: P2P social networking: early experiences and insights. *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*.

Catledge L. and Pitkow J.. Characterizing browsing behaviors on the World Wide Web. Computer Networks and ISDN Systems, 27(6), 1995.

Cooley, R., Mobasher, B and Srivastava, J. (1999) Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1).

Dakka, W., & Ipeirotis, P. G. (2008, April). *Automatic Extraction of Useful Facet Hierarchies from Text Databases*. Paper presented at the International Conference on Data Engineering, Cancún, México.

Deuze, M., Bruns, A., and Neuberger, C. (2007). Preparing for an age of participatory news. *Journalism Practice*, 1(3): 322-338.

Dong, Y., Tai, X., & Zhao, J. (2005). A Distributed Algorithm Based on Competitive Neural Network for Mining Frequent Patterns. *Proceedings of the International Conference on Neural Networks and Brain*, Beijing, China.

El-Hajj, M., & Zaiane, O. R. (2006, July). *Parallel leap: large-scale maximal pattern mining in a distributed environment*. Paper presented at the 12th International Conference on Parallel and Distributed Systems, Minneapolis, Minnesota, USA.

Ezeife, C.I., Lu, Y. and Liu, Y. (2005). PLWAP Sequential Mining. *Journal of Data Mining and Knowledge Discovery*, 10, 5-38.

Giannikopoulos, P., Varlamis, I. and Eirinaki, M. (2008). Mining frequent generalized patterns for Web personalization. *Proceedings of the workshop on Mining Social Data, 18th European conference on Artificial Intelligence*, Patras, Greece, 11-15.

Giannikopoulos, P., Varlamis, I. and Eirinaki, M. (2010). Mining frequent generalized patterns for Web personalization in the presence of taxonomies. *International Journal Data Warehousing and Mining* 6(1), 58-76

Han, J., Pei, J., Yin, Y. and Mao, R. (2004). Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Mining and Knowledge Discovery*, 8:53-87.

Heymann, P., & Garcia-Molina, H. (2006). *Collaborative Creation of Communal Hierarchical Taxonomies in Social Tagging Systems*. Technical Report 2006-10, Computer Science Department, InfoLab, Stanford University.

Idreos, S., Tryfonopoulos, C., Koubarakis, M. and Drougas, Y. (2004). Query Processing in Super-Peer Networks with Languages Based on Information Retrieval: the P2P-DIET Approach. In: *Proceedings of the international workshop on Peer-to-peer Computing and Databases*, Lecture Notes in Computer Science 3268, 496-505.

Javed, A., & Khokhar, A. (2004). Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, *16(3)*, 321-334.

Jelasity, M. and van Steen, M. (2002). *Large-Scale Newscast Computing on the Internet*. Internal report IR-503, Department of Computer Science, Vrije Universiteit.

Jiang, T. and Tan, A.H. (2006). Mining RDF Metadata for Generalized Association Rules. *Proceedings of the 17th international conference on Database and Expert Systems Applications*, Lecture Notes in Computer Science 4080:223-233.

Jiang, T., Tan, A-H. and Wang, K. (2007). Mining Generalized Associations of Semantic Relations from Textual Web Content. *IEEE Transactions on Knowledge and Data Engineering* 19(2):164-179.

Kantarcioglu, M. and Clifton, C. (2004). Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1026–1037.

Kirk, P. (2003). Gnutella Protocol Development. Retrieved June 27, 2011 from http://rfc-gnutella.sourceforge.net/

Li, H., Wang, Y., Zhang, D., Zhang, M. and Chang, E. (2008). PFP: Parallel FP-Growth for Query Recommendation. *Proceedings of the 2nd ACM conference on Recommender Systems*, Lausanne, Switzerland.

Li, L., Zhang, Ch., Wang, Y., Ji, Y. (2008b). Reliable and Scalable DHT-Based SIP Server Farm. *Proceedings of IEEE GLOBECOM 2008*, 1-6.

Maniatis, P. Roussopoulos, M., Giuli T.J., Rosenthal, D. and Baker, M. (2005). The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems* 23(1).

Napster website, (2011). Retrieved June 27, 2011 from http://www.napster.com.

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmer, M. and Risch, T. (2002). EDUTELLA: A P2P Networking Infrastructure Based on RDF. *Proceedings of the 11th International World Wide Web Conference.*

Pei, J., Han, J., Mortazavi-Asl, B. and Zhu, H. (2000). Mining access patterns efficiently from web logs. *Proceedings of the 4th Pacific-Asia conference on Knowledge Discovery and Data Mining*, Kyoto, Japan.

Poosala, V. (1997). *Histogram-based estimation techniques in database systems*. University of Wisconsin at Madison, Madison, Wisconsin.

Pouwelse, J. A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D. H. J., Reinders, M., van Steen, M. R. and Sips, H. J. (2007). TRIBLER: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20: 127–138. doi: 10.1002/cpe.1189

Pramudiono, I. and Kitsuregawa, M. (2003). Parallel FP-Growth on PC Cluster. *Proceedings of the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science 2637, 467-473.

Pramudiono, I. and Kitsuregawa, M. (2004) FPtax: Tree Structure Based Generalized Association Rule Mining. *Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Paris, France, 60-63

Pramudiono, I., Takahashi, K., Tung, A. and Kitsuregawa, M. (2005). Processing Load Prediction for Parallel FP-growth. *Proceedings of the Data Engineering workshop (DEWS)*, Saharu, Japan.

Qiong Gui, Xiao-hui Cheng (2009). A Privacy-Preserving Distributed Method for Mining Association Rules, *2009 International Conference on Artificial Intelligence and Computational Intelligence*, Shanghai, China.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S. (2001). A Scalable Content-addressable Network. *Proceedings of the conference on applications, technologies, architectures and protocols for computer communications*, San Diego, California

Said, A.M., Dominic, P.D.D. and Abdullah, A.B. (2009). A Comparative Study of FP-growth Variations. *International Journal of Computer Science and Network Security* 9(5), 266-272.

Sanderson, M., & Croft, W. B. (1999). Deriving concept hierarchies from text, *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 206–213.

Scholl, T., Bauer, B., Gufler, B., Kuntschke, R., Weber, D., Reiser, A. and Kemper, A. (2007). HiSbase: histogram-based P2P main memory data management. *Proceedings of the 33rd international conference on very large data bases (VLDB)*, 1394-1397.

Shintani, T., & Kitsuregawa, M. (1998, June). Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy. *Proceedings of the 24th ACM SIGMOD Conference in Management of Data*, Seattle, WA, USA.

Srikant, R. and Agrawal, R. (1995). Mining generalized association rules. *Proceedings of the 20th international conference on very large databases (VLDB)*, Zurich, Switzerland, 407-419

Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H. (2001) Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the Special Interest Group on Data Communication Conference*, New York, 149-160.

Su, C., & Sakurai, K. (2008). A Distributed Privacy-Preserving Association Rules Mining Scheme Using Frequent-Pattern Tree. In: C. Tang, C. X. Ling, X. Zhou, & N. J. Cercone & X. Li (Eds.), *Proceedings of the 4th International Conference on Advanced Data Mining,* Lecture Notes in Computer Science 5139, 170-181.

Sucahyo, Y.G., Gopalan, R.P., & Rudra, A. (2003). Efficiently Mining Frequent Patterns from Dense Datasets using a Cluster of Computers. *Proceedings of the 2003 Australian Conference on Artificial Intelligence*, 233-244.

Tanbeer S.K., Ahmed C.F., Jeong B. (2009). Parallel and Distributed Algorithms for Frequent Pattern Mining in Large Databases. *IETE Technical Review 26*, 55-65.

Tanenbaum, A.S. and van Steen, M. (2006). *Distributed Systems: Principles and Paradigms* (2nd ed), Prentice Hall, ISBN: 0132392275.

Tsai, F., Hana, W., Xua J. amd Chua, H. C. (2009). Design and development of a mobile peer-to-peer social networking application. *Expert Systems with Applications*, 36(8): 11077-11087.

Tsymbal, A. (2004). *The problem of concept drift: Definitions and related work*. Technical Report, Dublin, Ireland: Department of Computer Science, Trinity College, Dublin, Ireland.

Voulgaris, S., Jelasity, M. and van Steen, M. (2005). A Robust and Scalable Peer-to-Peer Gossiping Protocol. *Agents and Peer-to-Peer Computing*. Lecture Notes in Computer Science 2282:9-25.

Vu, Q.H.., Ooi, B.C., Rinard, M. and Tan, K.L. (2009). Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems. *Transactions on Data and Knowledge Engineering*. 21(4), 595-608.

Wang, J., Han, J. and Pei, J. (2003). CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. *Proceedings of the Special Interest Group on Knowledge Discovery in Data Conference*, Washington, DC, 236-245.

Wang, Y. and Dasgupta P. (2005). P2P Volunteers for Reliable Server Farms, *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*.

Yu, K-M., Zhou, J., & Hsiao, W.C. (2007). Load Balancing Approach Parallel Algorithm for Frequent Pattern Mining. *Proceedings of the Parallel Computing Technologies Conference*, Lecture Notes in Computer Science 4671:623-631.

Zaïane, O. R., El-Hajj, M., & Lu, P. (2001). *Fast Parallel Association Rule Mining Without Candidacy Generation*. *Proceedings of ICDM 2001*, 665-668.

Zaki, M. J., Ogihara, M., Parthasarathy, S., & Li, W. (1996). *Parallel Data Mining for Association Rules on Shared-memory Multi-processors* (Tech. Rep.). Rochester, USA: University of Rochester.