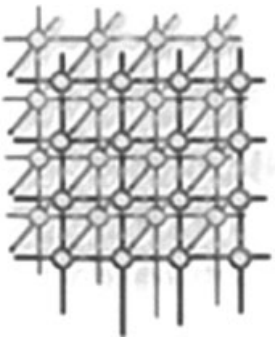
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:658–682
Published online 30 November 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1545



Mining@home: toward a public-resource computing framework for distributed data mining

C. Lucchese[1,*,†], C. Mastroianni[2], S. Orlando[3] and D. Talia[2,4]

[1]*I.S.T.I.-C.N.R.-HPC Laboratory, Via Moruzzi 1, Pisa 56100, Italy*
[2]*I.C.A.R., C.N.R., Rende, Italy*
[3]*Department of Computer Science, University of Venice, Italy*
[4]*D.E.I.S., University of Calabria, Rende, Italy*

SUMMARY

Several classes of scientific and commercial applications require the execution of a large number of independent tasks. One highly successful and low-cost mechanism for acquiring the necessary computing power for these applications is the 'public-resource computing', or 'desktop Grid' paradigm, which exploits the computational power of private computers. So far, this paradigm has not been applied to data mining applications for two main reasons. First, it is not straightforward to decompose a data mining algorithm into truly independent sub-tasks. Second, the large volume of the involved data makes it difficult to handle the communication costs of a parallel paradigm. This paper introduces a general framework for distributed data mining applications called *Mining@home*. In particular, we focus on one of the main data mining problems: the extraction of *closed frequent itemsets* from transactional databases. We show that it is possible to decompose this problem into independent tasks, which however need to share a large volume of the data. We thus introduce a *data-intensive computing network*, which adopts a P2P topology based on super peers with caching capabilities, aiming to support the dissemination of large amounts of information. Finally, we evaluate the execution of a pattern extraction task on such network.

Received 10 March 2009; Revised 3 August 2009; Accepted 16 August 2009

KEY WORDS: public-resource computing; desktop grids; data mining; closed frequent itemsets; peer-to-peer computing

*Correspondence to: C. Lucchese, I.S.T.I.-C.N.R.-HPC Laboratory, Via Moruzzi 1, Pisa 56100, Italy.
†E-mail: claudio.lucchese@isti.cnr.it

Contract/grant sponsor: European Commission; contract/grant numbers: IST-FP6-004265, IST-FP7-215483

1. INTRODUCTION

Today's information society has become, with no doubt, a restless producer of data of many kinds. A diverse amount of data is being collected: from traditional supermarket transactions, credit card records, phone calls records, census statistics, and GPS tracks to other less usual kinds of data, such as astronomical images, molecular structures, DNA sequences, and medical records. We are witnessing an exponential growth in the volume of the data being collected, which implies an exponential growth in the interest for analyzing and extracting useful knowledge from such data. The will and the need to benefit from this largely available data may find an answer in data mining.

Nowadays, data mining applications need to deal with increasingly larger amounts of data, so that, in the future, they will likely become large-scale and expensive data analysis activities. For this reason, the size of the data and the scalability of algorithms have always been central issues in the data mining community. While several distributed mining approaches for small-sized workstation networks have been designed, they can hardly keep up with such data growth. In fact, they have not been proved to scale to very large data sets (order of terabytes), and usually require expensive data transfers during every iteration of the mining.

In this work we aim to explore the opportunities offered by the *volunteer computing paradigm* for supporting the execution of costly data mining jobs that need to explore very large data sets. During the recent years, volunteer computing has become a success story for many scientific applications. In fact, Desktop Grids, in the form of volunteer computing systems, have become extremely popular as a means for exploiting huge amount of low-cost computational resources with a few manpower getting involved. BOINC [1] is by far the most popular volunteer computing platform available today, and to date, over 5 million participants have joined various BOINC projects. The core BOINC infrastructure is composed of a scheduling server and a number of clients installed on users' machines. The client software periodically contacts a centralized scheduling server to receive instructions for downloading and executing a job. After a client completes the given task, it uploads the resulting output files to the scheduling server and requests further work. BOINC was successfully used in projects such as Seti@home, Folding@home, and Einstein@home.

On the other hand, the nature of data mining applications is very different from the usual '@home' applications. First, they are not easily decomposable into a set of small independent tasks. Second, they are data-intensive, as any sub-task needs to work on a large portion of the data. These two issues make it very challenging to distribute sub-tasks to volunteer clients. In fact, no algorithm has been designed for propagating large amounts of data in a public computing framework. Nevertheless, we believe that data mining may take advantage of volunteer computing in order to accomplish complex tasks that would otherwise be intractable.

In this paper we present a general framework, called *Mining@home, that supports the distributed execution of data mining applications by exploiting volunteer computing networks.* In particular, we focus on the *closed frequent itemsets mining problem* (CFIM). This requires to extract a set of significant patterns from a transactional data set, among the ones occurring more frequently. We discuss its parallelization issues, and provide a deep analysis of a set of partitioning strategies addressing the load imbalance problem. We also introduce a novel *data-intensive computing network*, which is able to efficiently support our mining task by adopting a volunteer computing paradigm. The network exploits caching techniques across a super-peer network to leverage the cost of spreading large amounts of data to all the computing peers.

DOI: 10.1002/cpe

The paper is structured as follows: Section 2 summarizes the related work in the fields of parallel mining of frequent itemsets and large-scale distributed computing with the volunteer paradigm. Section 3 presents D-CLOSED, a distributed algorithm that efficiently solves the CFIM problem. Section 4 describes the *Mining@home* super-peer network on which the CFIM algorithm is executed and the protocol exploited to disseminate and cache data and run the CFIM jobs. This section also defines a number of different caching strategies that can be used depending on the capabilities of the available hosts. Section 5 evaluates the performance of the envisioned algorithm, with a particular focus on the scalability features, and compares the different caching strategies. Finally, Section 6 discusses achievements and the possible future enhancements. Section 7 concludes the paper.

2. RELATED WORK

Many algorithms and solutions have been devised for the discovery of *frequent itemsets*, yet targeting small-scale parallel and distributed environments.

In this work, we aim at moving two steps forward:

- First, we tackle the task of mining *closed frequent itemsets*, which are a meaningful subset of frequent itemsets (see Section 3). As regards the extraction of closed itemsets in parallel, there exists only a multi-threaded algorithm called MT-CLOSED [2].
- Second, we propose a *data-intensive computing network*, inspired by volunteer computing, which is able to handle the dynamic behavior of the available resources, and alleviates the cost of spreading large amount of data through the network of computing nodes (see Section 4).

In the following, we thus review some of the main proposals presented in the literature, regarding parallel pattern mining and large-scale distributed computing.

2.1. Parallel frequent pattern mining ...

In order to show the difficulties of data mining algorithms in processing large data sets, we report in Table I the running time and memory usage of the FP-CLOSE [3] algorithm, which is one of the fastest closed frequent itemsets mining (FIM) algorithms according to the FIMI 2003 contest [4]. We used 'medium-sized' data sets: Accidents (34 MB) and USCensus1990 (521 MB), and varied the minimum support threshold, that is the minimum number of occurrence that an itemset must have in order to be considered relevant. These experiments were run on an Intel Xeon 2 GHz box equipped with 1GB of physical memory. Even with the smallest data set, when decreasing the minimum support threshold, the memory size becomes an issue: either the process is slowed down to 7 hours of computation because of memory swaps, or it fails due to large memory requirements, exceeding the virtual memory capacity of 2 GB.

This behavior is typical of most pattern mining algorithms. When increasing the size of the database, such problems become apparent, since the memory occupation and the running time increase at least proportionally. As a result, state-of-the-art algorithms are able to run to completion only with large support thresholds, that is, extracting only a small fragment of the knowledge hidden inside the data set.

DOI: 10.1002/cpe

Table I. FP-CLOSE running time (in hh.mm:ss) and memory usage on the data sets Accidents and USCensus1990.

Data set					
Accidents			USCensus1990		
$\bar{\sigma}$ (%)	Time	Memory	$\bar{\sigma}$ (%)	Time	Memory
5.0	25:03	600 MB	45	12:09	452 MB
4.0	44:50	1023 MB	40	37:05	1273 MB
3.0	7.09:12	1225 MB	35	*failed after 1.15:31*	
2.0	*failed after 3.12:14*		30	*failed after 21:34*	

In this scenario, parallelism can be used to overcome the limits of a single machine. On the one hand, memory requirements can (possibly) be fulfilled by the aggregate memory of a pool of machines. On the other hand, the speed-up provided by a number of CPUs is very welcome in such data mining tasks, where the total computational cost can be measured in hours. Still, pattern mining algorithms cannot be trivially parallelized, and the mining of large data sets usually implies large communication and synchronization costs.

To the best of our knowledge, MT-CLOSED [2] is the only algorithm for mining in parallel *closed* frequent itemsets. However, MT-CLOSED is a multi-threaded algorithm designed for multi-core architectures, therefore using shared memory and having a limited degree of parallelism. Section 3 discusses the problem of mining closed frequent itemsets, and proposed a new distributed algorithm.

A very recent effort toward the *distributed* mining of 'terabyte-sized data sets' is [5]. The authors focus on the problem of discovering frequent itemsets, i.e. itemsets occurring more than a given threshold in a transactional database (see Section 3). First, they show that state-of-the-art algorithms for parallel pattern mining, such as DD [6], IDD, and HD [7], are not able to scale at all. Only CD [6] performs sufficiently well in terms of speed-up, but still, the total computation time is large compared with more efficient non-parallel implementations. Finally, they propose Distributed FP-Growth (DFP), an algorithm inspired by FP-GROWTH [8], which is one of the fastest sequential pattern mining algorithms. Not only does DFP have a small computational time, but it has a considerable speed-up of factor 5.24 when going from 8 to 48 processors (ideal speed-up value is 6). They use a baseline of 8 processors, because the mining task cannot be accomplished on a single machine. Note that in their work, the authors assume that the data set is already distributed across the 48 processors, therefore, DFP does not take into consideration any start-up communication to spread the data set among the various mining nodes. Moreover, the authors assume that the set of processing nodes is *static* and *continuously available*. This assumption may be restrictive when the cost of the processing increases: even within a small organization, new nodes may become available, or some nodes may provide their computing power only for a limited amount of time.

In [9], the authors use the Map-Reduce framework [10] to discover co-occurring items on a very large cluster with more than 2000 machines. This problem has lower complexity compared with frequent pattern mining, since a small portion of the frequent itemsets, consisting of very short patterns, is returned. However, the large data sets used and the very promising results obtained, encourage the research on mining algorithms in large distributed environments.

DOI: 10.1002/cpe

2.2. ... and large-scale distributed computing

The term 'volunteer computing' or 'public-resource computing' [11] is used for applications in which jobs are executed by privately owned and often donated computers that use their idle CPU time to support a given, usually scientific, computing project. The pioneer project in this realm is Seti@home [12], which has attracted millions of participants wishing to contribute to the digital processing of radio telescope data in the search for extra-terrestrial intelligence. A number of similar projects, such as Folding@home and Einstein@home, are supported today by the software infrastructure that evolved out of that project: the Berkeley Open Infrastructure for Network Computing, or BOINC [1]. The core BOINC infrastructure is composed of a scheduling server and a number of clients installed on users' machines. The BOINC middleware is especially well suited for CPU-intensive applications but is somewhat inappropriate for data-intensive tasks due to its centralized nature that currently requires all data to be served by a centrally maintained server.

XtremWeb [13,14] is another Desktop Grid project that, like BOINC, works well with 'embarrassingly parallel' applications that can be broken into many independent and autonomous tasks. XtremWeb follows a centralized architecture and uses a three-tier design consisting of a worker, a coordinator, and a client. The XtremWeb software allows multiple clients to submit task requests to the system. When these requests are dispensed to workers for execution, the workers will retrieve the necessary data and software to perform the analysis. The role of the third tier, called the coordinator, is to decouple clients from workers and to coordinate task executions on workers. Unfortunately this approach cannot support applications that are not 'embarrassingly parallel'.

Some recent efforts aim to exploit distributed architectures for mining tasks. In [15], a comparison is presented among four different strategies for the distribution of data to multiple nodes in a Cloud environment: 'streaming', 'pull', 'push', and 'hybrid'. The hybrid strategy combines the properties of pull and push algorithms, where data is first delivered to a subset of intermediate nodes, endowed with reliable and high-capacity storage devices, and then is requested by the nodes that perform the mining tasks. In this respect, the caching algorithm defined in *Mining@home* resembles, and further enhances, the hybrid strategy described in [15].

In [16], the results of the execution of a distributed algorithm for computing histograms from multiple high-volume data streams are presented. The tests were executed on a data mining middleware that uses Composible-UDT, a data transfer protocol specialized for high-bandwidth and long-haul connections. However, the middleware is specifically focused on the computation of histograms and may not be easily generalized to different data mining tasks. Moreover, experiments were performed with only a few nodes; therefore, the scalability issues could not be taken into account.

Grid Weka [17] and Weka4WS [18] are efforts aimed at exploiting Grid facilities and services to support distributed data mining algorithms. They extend the Weka toolkit to enable the use of multiple computational resources when performing data analysis. In these systems, a set of data mining tasks can be distributed across several machines in an *ad hoc* environment. These kinds of *ad hoc* environments limit the maximum number of new machines that may join the computation, and assume that no faults are possible.

Falcon [19] is a system tailored for distributed application *monitoring* and *steering*. Monitoring is achieved by instrumenting the application code with a collection of sensors. These sensors will transmit data to a central monitor during execution. The monitor provides user interfaces for the

DOI: 10.1002/cpe

online analysis of application performance by a human expert. Falcon, then allows the online steering (i.e. reconfiguration) either driven by the human expert or dynamically through policies provided by the user. The goal of steering is to improve the performance of the application, e.g. by improving the load balance. Even though Falcon is a very general tool, it does not address one of the major issues of large-scale data mining applications: the data dissemination. Falcon does not provide any support for a smart and efficient data distribution. Conversely, our data-intensive computing network provides a replication service that reduces the data transfer costs and the latencies.

In [20], a framework is presented that attempts to combine the strengths of a volunteer distributed computing approach such as BOINC with decentralized, yet secure and customizable, peer-to-peer data sharing practices. This approach differs from the centralized BOINC architecture in that it seeks to integrate P2P networking directly into the system, as job descriptions and input data is provided to a P2P network instead of directly to the client. The P2P-based public computing framework discussed in [20] was applied to the analysis of gravitational waveforms for the discovery of user-specified patterns that may correspond to 'binary stars'. However, that use case is simpler than the data mining problem discussed in this paper, because in the astrophysical application, input data is partitioned in disjoint subsets that are assigned to worker nodes, and jobs are all very similar in terms of space and time complexity.

Regarding data-dissemination, several solutions have been explored by content delivery services such as Akamai (www.akamai.com<http://www.akamai.com>). Akamai relies on the smart use of the DNS services in order to provide the URL of a copy of the requested object that is as close as possible to the user. Clearly, this approach is not the most effective in our scenario: first, permanent storage of data is not required, second, and more importantly, our data-intensive network allows for the replication of portions of data, accommodating the needs of the various sub-tasks.

3. PARALLEL MINING OF CLOSED FREQUENT ITEMSETS

FIM is a demanding task common to several important data mining applications that look for interesting patterns within databases (e.g. association rules, correlations, sequences, episodes, classifiers, clusters). The problem can be stated as follows.

Definition 1 (*FIM*: *Frequent Itemset Mining Problem*). Let $\mathscr{I} = \{x_1, \ldots, x_n\}$ be a set of distinct literals, called *items*. An *itemset* X is a subset of $\mathscr{I}$. If $|X| = k$, then X is called a *k-itemset*. A *transactional database* is a bag of itemsets $\mathscr{D} = \{t_1, \ldots, t_{|\mathscr{D}|}\}$ with $t_i \subseteq \mathscr{I}$, usually called *transactions*. The *support* of an itemset X in database $\mathscr{D}$, denoted $\sigma_{\mathscr{D}}(X)$ or simply $\sigma(X)$ when $\mathscr{D}$ is clear from the context, is the number of transactions that include X. Given a user-defined *minimum support* $\bar{\sigma}$, an itemset X such that $\sigma_{\mathscr{D}}(X) \geq \bar{\sigma}$ is called *frequent* or *large* (since they have large support). The *FIM Problem* requires to discover all the frequent itemsets in $\mathscr{D}$ given $\bar{\sigma}$.

We denote with $\mathscr{L}$ the collection of frequent itemsets, which is indeed a subset of the huge search space given by the power set of $\mathscr{I}$.

State-of-the-art FIM algorithms visit a lexicographical tree spanning over such search space (Figure 1), by alternating *candidate generation* and *support counting* steps. In the candidate generation step, given a frequent itemset X of $|X|$ elements, new candidate $(|X| + 1)$-itemsets Y are generated as supersets of X that follow X in the lexicographical order. During the counting step, the

DOI: 10.1002/cpe

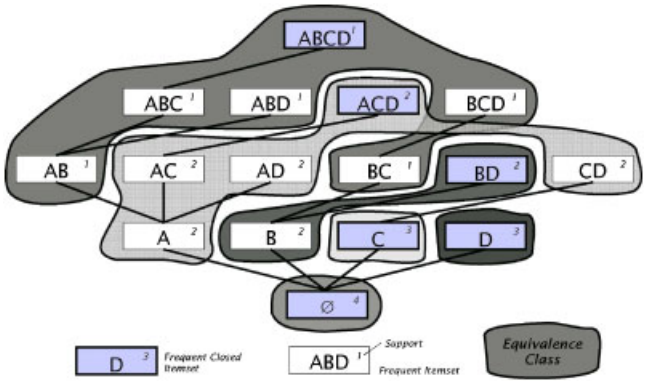


Figure 1. Lexicographic spanning tree of the frequent itemsets, with closed itemsets and their equivalence classes, mined with $\bar{\sigma} = 1$ from the data set $\mathscr{D} = \{\{B, D\}, \{A, B, C, D\}, \{A, C, D\}, \{C\}\}$.

support of such candidate itemsets is evaluated on the data set, and if some of those are found to be frequent, they are used to reiterate the algorithm recursively. Note that the aforementioned lexicographical order is typically based on the increasing frequency order of singletons. Such reordering of items is proved to reduce the search space, since infrequent itemsets are pruned early.

The collection of frequent itemsets $\mathscr{L}$ extracted from a data set is usually very large, with a size exponential w.r.t. the number of frequent literals. This makes the task of the analyst hard, since he has to extract useful knowledge from a huge amount of patterns, especially when very low minimum support thresholds are used. The set $\mathscr{C}$ of closed itemsets [21] is a concise and lossless representation of the frequent itemsets that has replaced traditional patterns in many other mining tasks, e.g. sequences [22], trees [23], graphs [24], etc. In the following section, we describe the algorithmic issues related to the discovery of closed frequent itemsets from databases.

3.1. CFIM: the closed FIM problem

We now introduce formally the problem of mining closed frequent itemsets. Let T be a set of transactions in $\mathscr{D}$ and I an itemset; the concept of closed itemset is based on the following two functions f and g:

$$f(T) = \{i \in \mathscr{I} \mid \forall t \in T, i \in t\}$$
$$g(I) = \{t \in \mathscr{D} \mid \forall i \in I, i \in t\}$$

Function f returns the set of items included in all the transactions in the set T, i.e. their intersection, whereas function g returns the set of transactions supporting a given itemset I. We can write $\sigma(I) = |g(I)|$.

Definition 2. An itemset I is said to be *closed* if and only if

$$c(I) = f(g(I)) = f \circ g(I) = I$$

where the composite function $c = f \circ g$ is called the *Galois operator* or *closure operator*.

DOI: 10.1002/cpe

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class *if and only if* they have the same closure. Equivalently, two itemsets belong to the same class *iff* they are supported by the same set of transactions. We call these partitions of the lattice *closure-based equivalence classes*.

We can also show that an itemset I is closed *iff* no superset of I with the same support exists. Therefore mining the *maximal* elements of all the closure-based equivalence classes corresponds to the mining all the closed itemsets.

Example 1. Figure 1 shows the lattice of frequent itemsets derived from the simple data set reported in the same figure, mined with $\bar{\sigma} = 1$. We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that closed itemsets (six) are remarkably less than frequent itemsets (sixteen).

Definition 3 (*CFIM: Closed FIM*). Let $\mathscr{D}$ be a transactional data set and $\bar{\sigma}$ a given minimum support threshold; the *Closed Frequent Itemset Mining Problem* requires to discover all the itemsets X such that $\sigma(X) \geq \bar{\sigma}$ (they are frequent) and $c(X) = X$ (they are closed).

CFIM algorithms are quite similar to FIM algorithms, with the addition of a *closure calculation* step: as soon as a new frequent itemset is found, its closure is computed and this is later used to reiterate the mining. Since there are several candidates for a given closed itemset, e.g. A and CD are candidates with the same closure ACD, some kind of *duplicate detection* technique must be exploited. State-of-the-art algorithms such as FP-CLOSE [3], CHARM [25], and CLOSET [26] exploit the Sub-Sumption lemma to detect useless candidates.

Lemma 1 (Sub-sumption Lemma). *Given an itemset X and closed itemset $Y = c(Y)$, if $X \subset Y$ and $\sigma(X) = \sigma(Y)$ then $c(X) = Y$. In this case we say that X is sub-sumed by Y.*

Proof 1. If $X \subset Y$, then $g(Y) \subseteq g(X)$. Since $\sigma(X) = \sigma(Y) \Rightarrow |g(Y)| = |g(X)|$, then $g(Y) = g(X)$. Finally, $g(X) = g(Y) \Rightarrow f(g(X)) = f(g(Y)) \Rightarrow c(X) = c(Y)$. □

As soon as a closed itemset is discovered, it is stored in an incremental data structure. This *historical collection* is used to understand whether an itemset is closed or not. Given the itemset X, if there exists an already mined closed itemset Y that includes X and that has the same support of X, then X is not closed and its closure $c(Y)$ was already discovered.

According to Lemma 1, we can assess the closed-ness of an itemset by maintaining a *global knowledge of the collection of closed itemsets*. The size of this collection grows exponentially when decreasing the minimum support threshold, which makes this technique expensive both in time and space. In time because it requires the possibly huge set of closed itemsets mined so far to be searched for the set-inclusions. In space, because, in order to efficiently perform set-inclusion checks, all the closed itemsets have to be kept in the main memory.

In a way, the need to maintain a global and evolving data structure does not comply with the general requirements of a *divide et impera* approach needed for an efficient parallelization. Instead of having a pool of independent sub-tasks, a CFIM algorithm is decomposed in *dependent* sub-problems that actually co-operate and communicate, meaning that the output of one task, i.e. the

DOI: 10.1002/cpe

extracted closed frequent itemsets, is the input of the next task, i.e. the historical collection to be used for duplicate detection.

In the following, we introduce D-CLOSED, an algorithm inspired to DCI-CLOSED [27] and MT-CLOSED [2], for mining closed frequent itemsets in a distributed setting.

3.2. D-CLOSED: the first distributed algorithm for the CFIM problem

The first two passes of the algorithm are devoted to the construction of the internal representation of the data set. A first scan is performed in order to discover the set of frequent singletons $\mathscr{L}_1$. Then, a second scan is needed to create a vertical bitmap BM of the data set. For each item $i \in \mathscr{L}_1$, a $0-1$ array of bits is stored, where the j-th bit is set if the item i is present in the transaction j. The resulting bitmap has size $|\mathscr{L}^1| \times |\mathscr{D}|$ bits. Note that tid-lists are stored contiguously in increasing frequency order, i.e. the first row of the bitmap contains the tid-list of the least frequent item.

The kernel of the algorithm consists in a recursive procedure that exhaustively explores a sub-tree of the search space given its root. The input of this procedure is a seed closed itemset X and its tid-list $g(X)$. Similar to other CFIM algorithms, given a closed itemset X, new *candidates* $Y = X \cup i$ are created according to the lexicographic order. If a candidate Y is found to be frequent, then its closure is computed and $c(Y)$ is used to continue the recursive traversal of the search space. In order to detect duplicates, D-CLOSED introduces the concept of pro-order and anti-order set:

Definition 4 (*Pro-order and Anti-order sets*). Let $\prec$ be an order relation among the set of literals in the data set, and let $Y = X \cup i$ be a generator. We denote with Y^+ the *pro-order* and with Y^- the *anti-order* set of items for Y defined as follows:

$$Y^- = \{j \in (\mathscr{I} \setminus Y) \mid j \prec i\}$$
$$Y^+ = \{j \in (\mathscr{I} \setminus Y) \mid i \prec j\}$$

It is easy to show that an algorithm that discards all those candidates Y for which $c(Y) \cap Y^- \neq \emptyset$ is correct (see [2] for a complete proof). The rationale is that for each closed itemset Z there is a candidate Y that needs only the items in Y^+ to calculate the closure $c(Y) = Z$, and this closure is in accordance with the underlying lexicographic tree, which spans the whole lattice. Conversely, a candidate that needs items in Y^- is redundant.

Example 2. With reference to Figure 1, suppose that the core recursive mining procedure is invoked on the candidate $Y = \emptyset \cup B$. By definition we have that $Y^- = \{A\}$ and $Y^+ = \{C, D\}$.

The closure of Y is $c(Y) = \{B, D\}$, and it is not discarded since it has empty intersection with the anti-order set Y^-. A new item is taken from the pro-order set and a new candidate $Y' = \{B, D\} \cup C$ is used to start a new recursion of the mining with $Y'^- = \{A\}$ and $Y'^+ = \emptyset$.

The closure of Y' is $C(Y') = \{A, B, C, D\}$. It has non-empty intersection with the anti-order set Y'^-, and is therefore discarded as being redundant. Indeed, the itemset $\{A, B, C, D\}$ was already discovered after exploring the sub-tree rooted in A.

DOI: 10.1002/cpe

To understand which items are needed to calculate the closure of a given itemset, it is possible to exploit the following Extension Lemma:

Lemma 2 (Extension Lemma). *Given an itemset X and an item $i \in \mathcal{I}$, if the set of transactions supporting X is a subset of the transactions supporting i, then i belongs to the closure of X, and vice versa, i.e. $g(X) \subseteq g(i) \Leftrightarrow i \in c(X)$.*

Proof 2. See [27]. □

Note that the check $g(X) \subseteq g(i)$ can be performed very efficiently given the adopted bit-vector representation. In fact, it is sufficient to perform a bitwise inclusion between the tid-lists of X and i.

Every single closed itemset X can be thought as the root of a sub-tree of the search space that can be mined independently of any other (non-overlapping) portion of the search space. Note that this

Algorithm 1 D-CLOSED algorithm.

```
1:  function D-CLOSED (𝒟, σ̄)
2:      𝒞 ← ∅                                          ▷ the collection of frequent closed itemsets
3:      𝓛₁ ← get-frequent-singletons(𝒟, σ̄)             ▷ first scan
4:      BM ← vertical-bitmap(𝒟, 𝓛₁)                    ▷ second scan
5:      𝒥 ← JOBSCREATE()                               ▷ create independent jobs
6:      for all J ∈ 𝒥 do                               ▷ jobs can be executed in parallel
7:          MINE-NODE(J.X, J.g(X), J.X⁻, J.X⁺, 𝒞, σ̄, BM)
8:      end for
9:      return 𝒞
10: end function

11: procedure MINE-NODE (X, g(X), X⁻, X⁺, 𝒞, σ̄, BM)
12:     c(X) ← X ∪ {j ∈ X⁺ | g(X) ⊆ g(j)}              ▷ compute closure
13:     𝒞 ← 𝒞 ∪ c(X)
14:     Y⁻ ← X⁻                                        ▷ new anti-order set
15:     Y⁺ ← X⁺ \ c(X)                                 ▷ new pro-order set
16:     while Y⁺ ≠ ∅ do
17:         i ← pop_min_≺(Y⁺)                          ▷ select and remove an item from Y⁺
18:         Y ← c(X) ∪ i                               ▷ candidate generation
19:         g(Y) = g(X) ∩ g(i)
20:         if |g(Y)| ≥ σ̄ then                         ▷ frequency check
21:             if ¬∃j ∈ Y⁻ | g(Y) ⊆ g(j) then         ▷ order-preservation check
22:                 MINE-NODE(Y,g(Y),Y⁻,Y⁺, 𝒞, σ̄, BM)  ▷ recursive visit
23:                 Y⁻ ← Y⁻ ∪ i
24:             end if
25:         end if
26:     end while
27: end procedure
```


DOI: 10.1002/cpe

is an original feature of D-CLOSED, since other algorithms need to share the collection of closed itemsets discovered so far. Conversely, D-CLOSED needs parallel sub-tasks to share the *static* vertical bitmap representing the data set. In addition, thanks to the use of pro-order and anti-order sets, it is possible to reduce the amount of shared data, since only the tid-lists of the items belonging to either of those sets are of interest. In addition, in [2] it is shown how to prune such sets during the mining.

Thus, it is possible to partition the whole mining task into independent regions, i.e. sub-trees of the search space, each of them described by a distinct *job descriptor* $J = \langle X, g(X), X^-, X^+ \rangle$. The job descriptor J suffices to identify a given sub-task, or *job*, of the mining process. In principle, we could split the entire search space into a set of disjoint regions identified by $J_1, \ldots, J_m$, and use some policy to assign these jobs to a pool of CPUs. Moreover, since the computation of each J_i does not require any co-operation with other jobs, and does not depend on any data produced by them (e.g. the historical closed itemsets), each job can be executed in a completely independent manner.

To summarize, we report the pseudocode of D-CLOSED in Algorithm 1. First the data set is scanned to discover frequent singletons, and then a vertical bitmap is built (lines 3–4). A collection of jobs is generated (line 5) and each executed independently (line 7). The core mining process immediately computes the closure of the given candidate itemset by exploring the pro-order set (line 12). Then, the remaining items in the pro-order set are used to generate new candidates (line 18), which, if order preserving (line 21), are used to reiterate the mining process (line 22). For the sake of clarity, in the above pseudocode, we reported that the whole bitmap *BM* is given in input to the MINE-NODE procedure, however, only a portion of it is necessary. Indeed, if $i \notin X^+$ and $i \notin X^-$, then the row of *BM* corresponding to $g(i)$ will never be accessed. We exploit this feature in one of the data dissemination strategies illustrated in Section 4.2.

In the following, we will see how to define and distribute the collection of jobs $\mathcal{J} = \{J_1, \ldots, J_m\}$.

3.3. Workload partitioning

One easy strategy would be to partition the frequent single items and assign the corresponding jobs to the pool of workers.

Definition 5 (*Naïve Lattice Partitioning based on* $\mathcal{L}_1$ (1P)). Let $\mathcal{L}_1 = \{i_1, \ldots, i_{|\mathcal{J}|}\}$ be the set of frequent items in the transactional data set $\mathcal{D}$. The lattice of frequent itemsets can be partitioned into $|\mathcal{L}_1|$ non-overlapping sets: the itemsets starting with i_1, the itemsets starting with i_2, etc. Each one is identified by a *job descriptor* defined as follows: $J_h = \langle X = \{i_h\}, g(X) = g(i_h), X^- = \{i \in \mathcal{L}_1 \mid i \prec i_h\}, X^+ = \{i \in \mathcal{L}_1 \mid i \not\preceq i_h\} \rangle$, for each $i_h \in \mathcal{L}_1$.

In Figure 2 we report the cost in seconds of every job resulting from the 1P partitioning. It is easy to see that while only a few jobs are very expensive, most of the remaining are two orders of magnitude cheaper. In fact, the 20 most expensive jobs, out of about 120, encompass more that 80% of the whole workload, as it can be seen in the middle plot of Figure 2, where jobs are sorted in the decreasing order of their cost. This results in a set of jobs leading to a significant load imbalance. Under the non-realistic assumption that the cost of each job is known in advance, we show the best

DOI: 10.1002/cpe

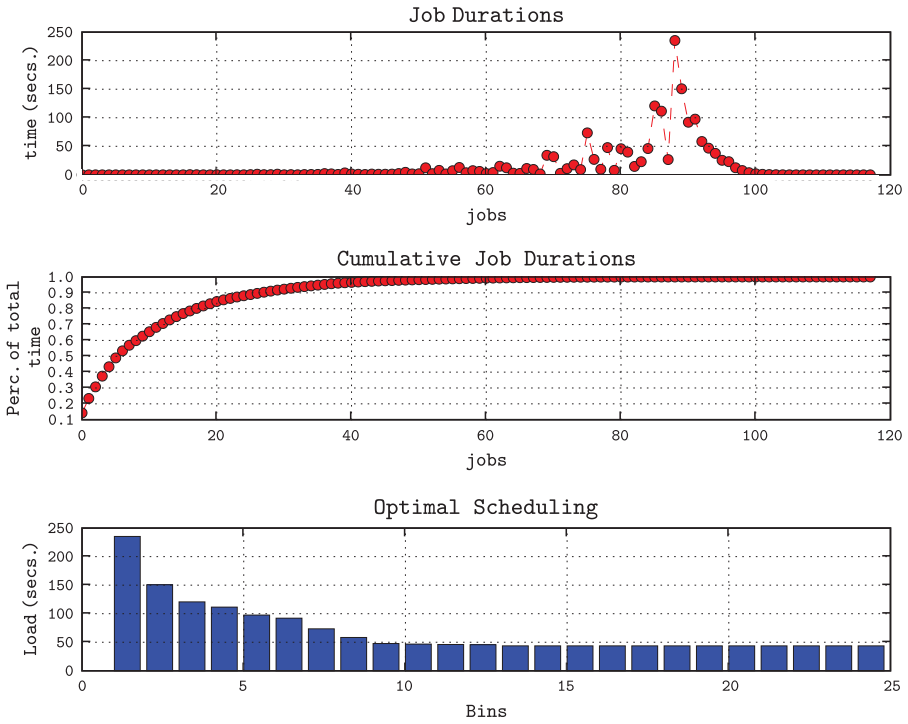


Figure 2. Job durations after 1P Partitioning on the Accidents data set with $\bar{\sigma} = 3\%$. In the top plot, jobs are reported in lexicographical order. In the middle plot, jobs are in decreasing cost order.

scheduling on a set of 24 machines: the result is a completely unbalanced computation, where the ratio between the minimum and the maximum load is 0.18. In particular, the most expensive job, which lasts about 250 s, overwhelms one node, and the other jobs are too cheap to feed the rest of the nodes.

Among the many approaches to solve this problem, an interesting one is [28]. The rationale behind the solution proposed by the authors is that there are two ways to improve the naïve scheduling described above. One option is to estimate the mining time of every single job, in order to assign the heaviest jobs first, and later the smaller ones to balance the load during the final stages of the algorithm. The other is to produce a much larger number of jobs thus providing a finer partitioning of the search space: having more jobs provides more degrees of freedom for the job assignment strategy. Their solution merges these two objectives in a single strategy. First, the cost of the jobs associated with the frequent singletons is estimated by running a mining algorithm on significant (but small) samples of the data set. Then, the most expensive jobs are split on the basis of the frequent 2-itemsets that they contain.

In our setting, we are willing to address very large data sets. In this case, the large number of the resulting samplings to be performed and their costs make the above strategy unsuitable. Therefore,

DOI: 10.1002/cpe

our choice is to avoid any expensive pre-processing, and to materialize jobs on the basis of the 2-itemsets in the Cartesian product $\mathscr{L}^1 \times \mathscr{L}^1$ as follows:

Definition 6 (*Lattice Partitioning based on 2-itemset* (2P)). Let $\mathscr{L}_1 = \{i_1, \ldots, i_{|\mathscr{L}_1|}\}$ be the set of frequent items in the transactional data set $\mathscr{D}$. The lattice of frequent itemsets can be partitioned into $\binom{|\mathscr{L}_1|}{2}$ non-overlapping sets: the itemsets starting with i_1, i_2, the itemsets starting with i_1, i_3, etc. Each is identified by a *job descriptor* defined as follows: $J_{hk} = \langle X = \{i_h, i_k\}, g(i_h) \cap g(i_k), X^- = \{i \in \mathscr{L}_1 \setminus i_h \mid i \prec i_k\}, X^+ = \{i \in \mathscr{L}_1 \mid i \not\succeq i_k\}\rangle$, for each $h < k$ and $i_h, i_k \in \mathscr{L}_1$.

This fine-grained partitioning produces a large number of jobs and sufficient degrees of freedom to evenly balance the load among workers. But still this may not be sufficient to achieve a good balancing of the various computing nodes. In Figure 3 we report some results coming from the 2P partitioning. As expected the number of jobs is much larger than before, i.e. ~ 7000 versus ~ 120. But also in this case, a few of them are significantly more expensive, whereas a large majority has a cost pretty close to zero. Also in this case, an omniscient scheduler, which knows in advance the cost of every single job, cannot provide an optimal scheduling. In the case of 24 computing nodes, we would have a ratio between the minimum and the maximum load of 0.8. We must also consider that such a large number of jobs may produce large communication overheads.

In order to overcome such problems, and to handle the dynamicity of a large computing network, we propose a novel partitioning strategy:

Definition 7 (*Dynamic Partitioning* (DP)). Let J^* be the a 2P partitioning of the workload. A DP is obtained in two steps. Before the mining, consecutive jobs in J^* that share the same prefix are grouped together in sets of size at most k. At mining time, every single job can be split into multiple parts by creating new jobs descriptors.

The first idea behind the DP strategy is to group together jobs when their amount is excessive, thus reducing the associated overheads. The maximum size k of a group could also be decided at run-time, depending on the number of jobs and nodes.

Notice that we group together two consecutive jobs only if they share the same prefix. For instance, $\{AB\}$ and $\{AC\}$ may be grouped together, whereas $\{AZ\}$ and $\{BC\}$ may not. The reason for this constraint is given by the *partitioning optimizations* usually adopted in the mining algorithm that we still want to use. Suppose that a job corresponds to the mining of all the itemsets beginning with a given item i: then any transaction that does not contain i can safely be disregarded. This technique significantly reduces the amount of data to be processed by a single job. This also explains why we only group 2-itemsets having the same prefix: we group jobs together only if they share the same projection of the data.

This data projection approach is very important in our framework. We can reduce the amount of data needed to accomplish a given job, and therefore the amount of data to be sent through the network.

The second idea is to take advantage of the expressive power of the job descriptor of our mining algorithm. In fact, every region of the search space can be expressed with a set of job descriptors, each of them resulting in an independent job. Therefore, a node that realizes that its load is too large, may return part of it to the computing network, by pushing a new job descriptor, and this

DOI: 10.1002/cpe

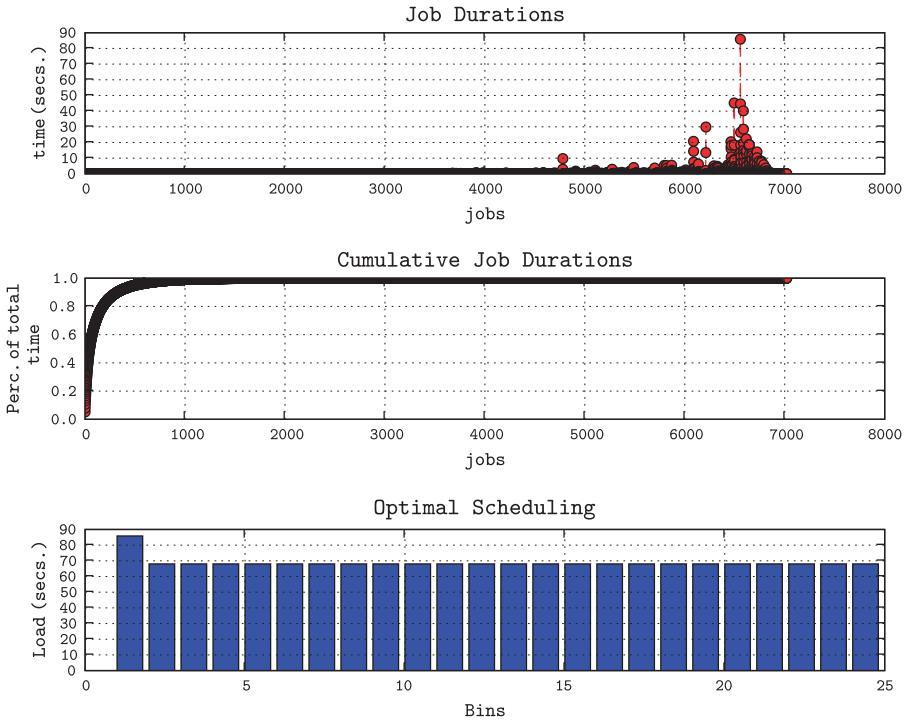


Figure 3. Job durations after 2P partitioning Accidents data set with $\bar{\sigma} = 3\%$. In the top plot, jobs are in lexicographical order. In the middle plot, jobs are in decreasing cost order.

operation can be repeated over time. Also, a node that wishes to leave the network may push the set of job descriptors corresponding to the part of the job not yet completed.

This partitioning strategy may thus solve every problem related to the workload imbalance or to the dynamicity of the network, resulting in a dynamic set of job descriptors evolving and adapting to the system over time.

4. *MINING@HOME*: A FRAMEWORK FOR DISTRIBUTED DATA-INTENSIVE APPLICATIONS

A preliminary framework of a data dissemination network was proposed in [20] for the processing of astronomical waveforms and in [29] for the analysis of audio files. In those scenarios, the partition of a large application into independent jobs is trivial as well as the distribution of input data to workers, and job balancing issues are hardly present since jobs are all very similar in terms of space and time complexity.

Here, we significantly extend and improve such a framework by proposing *Mining@home*: a new *data-intensive computing network* able to support D-CLOSED, the first distributed algorithm for the CFIM data mining problem.

DOI: 10.1002/cpe

4.1. A data-intensive computing network

In our network, we distinguish between nodes accomplishing the mining task and nodes supporting the data dissemination. Among the first we define:

- the *Data Source* is the node that stores the entire data set that must be analyzed and mined.
- the *Job Manager* is the node in charge of decomposing the overall data mining application in a set of independent tasks, according to the D-CLOSED algorithm. This node produces a *job advert* document for every task, which describes its characteristics and specifies the portion of the data needed to complete the task. The job advert actually corresponds to the job descriptor defined in Section 3.2. The job manager is also responsible for the collection of output results.
- the *Miners* are the nodes available for job execution. A miner first issues a *job query* to retrieve a job advert, then a *data query* to obtain the input data for that job.

Such a network can easily support massively parallel tasks as in traditional volunteer computing. The job manager takes care of supervising the overall process by assigning the various jobs to a dynamic set of miners. Unfortunately, this would not be sufficient to provide an efficient support to data mining tasks. Owing to the large volume of data, some effective strategy for its dissemination must be devised.

We propose a *data-intensive computing network*, which exploits the presence of a network of super-peers for the assignment and execution of jobs, and adopts caching strategies to improve the data delivery process. Specifically, the computing network exploits the presence of:

- *Super-Peer* nodes, which constitute the backbone of the network. Miners connect directly to a super-peer, and super-peers are connected with one another through a high-level P2P network.
- *Data-Cachers* nodes, which are super-peers having the additional ability to cache data and the associated data adverts. Data cachers can retrieve data from the data source or other data cachers, and later provide such data to miners.

The super-peer paradigm is chosen to let the system support several public computing applications concurrently, without requiring that every worker/miner to know the location of the job manager and/or of the data cachers. The super-peer paradigm allows the job and data queries issued by miners to rapidly explore the network and discover matching job and data adverts. In this kind of network it has to know the address of a neighbor super-peer only, irrespective of the application for which it wants to contribute.

The algorithm works as follows: a set of job adverts are generated by the job manager. Each job advert specifies the items that must be included in the frequent itemsets to be mined. An available miner issues a *job query* to retrieve one of these job adverts. Job queries can be delivered directly to the job manager; if the location of this node is not known (for example, if several data mining applications are concurrently running), they can travel the network through the super-peer interconnections. When a job advert is found that matches the job query, the related job is assigned to the requesting miner. The miner is also informed, through the job advert, about the data that it needs to execute the job. The required input data can be the entire data set stored in the data source, or a subset of it.

The miner does not download data directly by the data source, but issues a *data query* to discover a data cacher. This query can actually discover several data cachers: each of these sends an ack

DOI: 10.1002/cpe

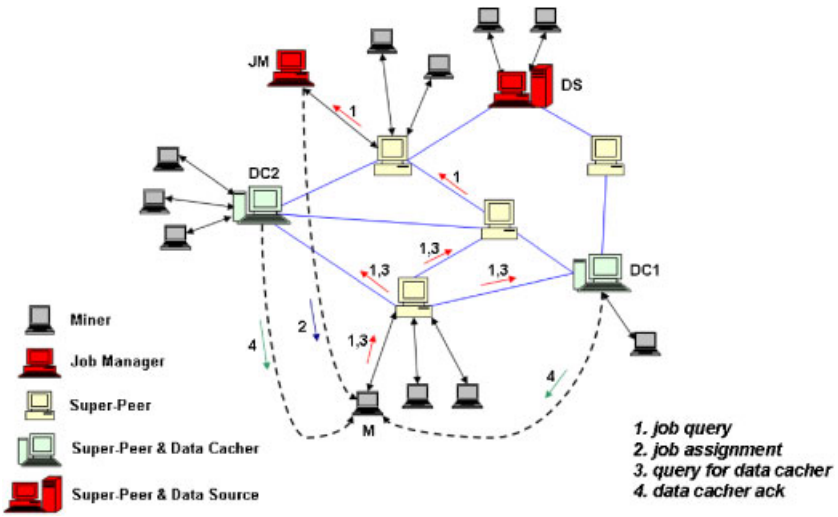


Figure 4. Caching algorithm in a sample super-peer network (1 of 2).

to the miner. Then the miner selects the nearest data cacher (or the most convenient data cacher according to some given strategy) and delegates to it the responsibility of retrieving the input data set from the data source, or from another data cacher that has already downloaded the data. After retrieving the input data, the cacher stores it, then passes it to the miner for job execution. The data cacher retrieves all the data set, even if the miner needs only a portion of it, in order to have the data available for other miners' requests.

The algorithm is illustrated in Figures 4 and 5, which show the messages exchanged in a network having seven super-peers (among which one is the data source and two are data cachers), a job manager, and several miners. In the first figure, the miner M, available to execute a job, issues a job query (step 1) that travels across the super-peer interconnections and gets to the job manager JM. The job manager assigns a job to the miner and sends it the job advert that describes the job and the input data required for its execution (step 2). Afterwards, the miner issues a query to discover a close data cacher (step 3) and in this case discovers two data cachers, which advertise their presence with an ack message (step 4). The subsequent steps of the algorithm are depicted in Figure 5. The miner selects the most convenient data cacher, in this case the data cacher DC_1 (step 5). Since DC_1 does not hold the required data, it issues a query to retrieve the data from the data source or from another data cacher (step 6). In this scenario, data is found in the data source DS. Hence, DC_1 retrieves data from DS (step 7), stores it in the cache in order to serve future miners requests, and provides it to the miner (step 8). The miner can now execute the job, and sends the output to the job manager, through a message not shown in the figure.

The algorithm includes a number of techniques that can make execution faster, depending on the state of the network and the dissemination of data. For example, in the case that the cacher DC_1 already owns the needed data, steps 6 and 7 are unnecessary. Moreover, a miner can exploit the results of discovery operations executed previously. Specifically, it needs to issue a query to

DOI: 10.1002/cpe

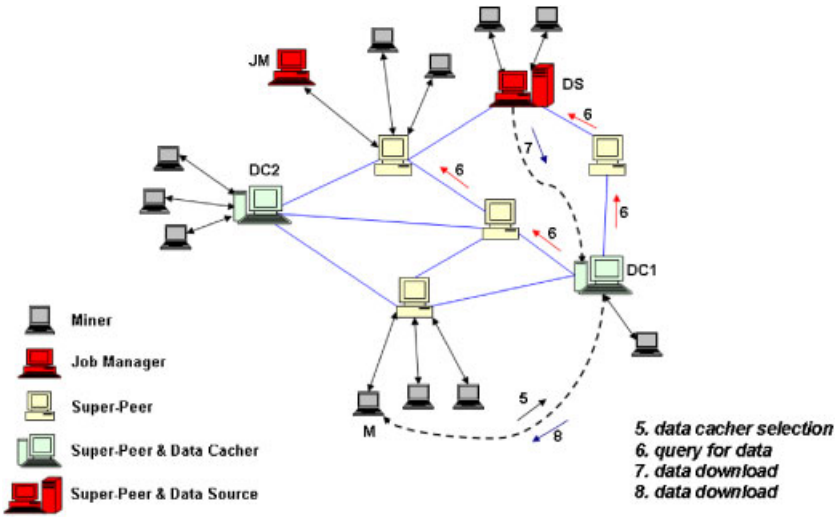


Figure 5. Caching algorithm in a sample super-peer network (2 of 2).

discover the job manager or the data cacher only the first time that it asks to execute a job. In the subsequent times the miner can decide to directly contact the job manager and the data cacher that it previously discovered. A miner could also have the ability to cache the data itself. This aspect is discussed in the following subsection.

The decentralized architecture is suitable to efficiently cope with node failures, which may ungracefully cause job terminations. The adopted strategy for failure management is the following: when a job is assigned to a miner, the local super-peer, which is assumed to be significantly more reliable than regular nodes, stores the assigned job advert and periodically checks the status of the miner. Whenever the super-peer detects an abrupt failure of the miner, it resumes the job advert and returns it to the job manager, so that the job can be assigned to another miner. This strategy also implies that each miner communicates the termination of jobs only to the local super-peer. Notice that this strategy allows for a scalable management of the workers, since it only implies sporadic communications among each super-peer and the miners directly connected to it.

4.2. Data and workload dissemination strategies

The presence of data cachers helps the dissemination of data and can improve the performance of the network. It is also useful to verify whether the miners themselves could give a contribution to speed up the computation, in case they also have the capability to store some input data (in general, the public-resource computing paradigm does not require hosts to store data after the execution of a job). In fact, it often happens that the input data of a job overlaps, completely or partially, with the input data of another job executed previously. Therefore, the miner could retrieve the whole data set when executing the first job, and avoid to issue a data query for the subsequent job.

DOI: 10.1002/cpe

Therefore, two different caching strategies have been analyzed and compared:

- *Strategy M-D*: *Minimum Download.* The miner downloads from the data cacher only the portion of the data set that it strictly needs for job execution, and discards this data after the execution.
- *Strategy F-D*: *Full Download.* The miner downloads from the data cacher the entire data set the first time that it has to execute a job. Even though the miner will only use a portion of the data set, data will be stored locally and can be used for successive job executions.

As mentioned at the end of Section 3, a miner can execute only a part of the assigned job if it is too large and resource consuming. Therefore, to support load balance in the network, we propose and evaluate a third strategy:

- *Strategy D-A*: *Dynamic Assignment.* A miner executes the job only for a limited amount of time. If this time expires and the job is not completed, the miner builds a new job advert that embraces the unexecuted part of the job, and sends this description back to the job manager, which will reassign it to another miner.

Note that the D-A strategy implies that the miner does not try to predict the length of the job before executing it. Indeed, such a prediction is very difficult and often erroneous [28], and it would require an extra time that may not be counterbalanced by the time saving obtained with the prediction. The D-A strategy can be combined both with the M-D and the F-D strategy.

Depending on the application, these simple strategies may be further improved. One possible approach could be to use the information present in the job adverts, in order to retrieve only those transactions of the data set that the miner does not already store. Indeed, a wide range of opportunities is open. We discuss them in Section 6.

5. PERFORMANCE EVALUATION

We used an event-based simulation framework (similar to that used in [29]) to analyze the performance of our super-peer protocol. In the simulation, the running times of the jobs were obtained by actually executing the MINE-NODE procedure for each available job, and measuring the elapsed times. To model a network topology that approximates a real P2P network as much as possible, we exploited the well-known power-law algorithm defined by Albert and Barabasi [30]. This model incorporates the characteristic of preferential attachment that was proved to exist widely in real networks.

The simulation scenario is summarized in Table II. The network contains 25 super-peers and 250 potential miners, randomly distributed among super-peers. The bandwidth and latency between two adjacent super-peers were set to 1 Mbps and 100 ms, respectively, whereas the analogous values for the connections among a super-peer and a local miner were set to 10 Mbps and 10 ms. If during the simulation a node (e.g. a data source or a data cacher) needs to simultaneously serve multiple communications (with different miners), the bandwidth of each communication is obtained by dividing the downstream bandwidth of the server by the number of simultaneous connections.

The largest transactional data set usually adopted is `WebDocs` [31], which is about 1 GB large. This is an inverted file of a large web collection, and it can be used to detect plagiarism or

DOI: 10.1002/cpe

Table II. Simulation scenario.

Scenario feature	Value
Number of super-peers, N_{sp}	25
Number of data cachers, N_{dc}	0–23
Number of available miners, N_{miners}	1–250
Average number of neighbors of a super-peer	4
Size of the input data file	300 Mb
Number of jobs, N_{job}	846 300
Latency between two adjacent super-peers	100 ms
Latency between a super-peer and a local worker	10 ms
Bandwidth between two adjacent super-peers	1 Mbps
Bandwidth between a super-peer and a local worker	10 Mbps

near-duplicate documents. We used the well-known IBM data set generator[‡] to create a data set with similar characteristics, but twice as large. The resulting data set is called `Synth2GB`: it has about 1.3 millions transactions and 2.5 thousands distinct items, for a total size of 2 GB. By running the algorithm with a minimum absolute support threshold of 30 000, we obtained statistics of about 846 300 jobs resulting from the 2P partitioning strategy described in Section 3.3, which were later grouped in 100s to reduce the total number of jobs. The machine used for this experiment is an Intel Xeon 2.00 GHz with 4 MB L2 cache and 4 GB main memory. We measured the execution times of every single job and used them in the simulation: each job assigned by the job manager was given the same characteristics (size of input data to download, execution time) as those of the real jobs executed in the serial experiment. To better simulate a realistic scenario, the unavoidable heterogeneity of a distributed environment was taken into account. Hence, the execution time of each job was multiplied by a factor randomly chosen in the interval [0.25, 4], in order to simulate the different computing powers of the available miners in the distributed environment. In the first experiments, the nodes were assumed to be completely reliable: the impact of node failures is examined later.

Figure 6 shows the overall running time needed to execute the jobs, by using strategies M-D and F-D, in the case that 200 miners are available, by varying the number of data cachers from 0 (meaning that data can only be retrieved by the data source) to 23 (in this case, every super-peer is also a data cacher, except the data source and the super-peer directly connected to the job manager). The time needed to complete the mining on a single machine, by executing all the jobs in sequence, and taking into account the multiplication by a random factor mentioned before, is 2060 h, about 85 days. Conversely, the distributed execution of the jobs with the algorithm presented in this paper allows the overall computation time to be reduced to 64 h with strategy M-D and 18 h with strategy F-D, with reduction factors of 32 and 114, respectively. The better performance of the second strategy is therefore confirmed, but of course this improvement should be counterbalanced by the requirement that miners allow data to be cached in their local memory.

[‡]Available at: `http://pomino.isti.cnr.it/~claudio/assoc_gen_linux.tgz`.

DOI: 10.1002/cpe

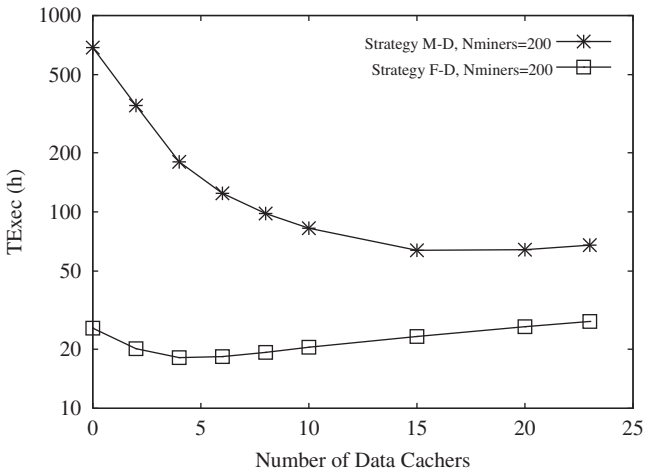


Figure 6. Overall execution time vs the number of data cachers with strategies M-D and F-D. The number of active miners is 200.

We notice that the number of available data cachers has an important influence on the overall execution time. With strategy M-D, the execution time decreases as the number of data cachers increases from 1 to about 15, since miners can concurrently retrieve data from multiple data cachers, thus reducing the duration of single download operations. However, the execution time increases when more than 15 data cachers are made available. To explain this, it must be considered that data cachers may retrieve data in parallel from the data source, so that the downstream bandwidth of the data source is shared among a large number of connections. The results show that the time needed to distribute data to more than 15 data cachers is not compensated by the time saved in data transfers from data cachers to miners. Therefore, an 'optimum' number of data cachers can be estimated. This number is 15 in this case, but in general depends on the application scenario, for example, on the number and the length of the jobs to execute. With strategy F-D, on the other hand, it is seen that the optimal number of data cachers is 4; in fact, a sort of second-level caching is operated directly by miners, hence there is less advantage to have nodes, i.e. the data cachers, which are specifically dedicated to the caching of data.

To analyze the scalability characteristics of the distributed algorithm, a number of experiments were performed by varying the number of available miners. Strategies M-D and F-D were compared: the number of data cachers was, respectively, set to 15 and 4, the values that were found to be optimal with the two strategies. The results, reported in Figure 7, confirm the clear advantage obtained with the strategy F-D, with any number of available miners. However, the scalability behavior is quite different in the two cases: with strategy M-D, the execution time decreases as the number of available miners is increased by up to about 50, but beyond this value the curve saturates, and therefore no advantage is granted by adding more miners. Conversely, strategy F-D can profitably exploit the presence of a much larger number of miners. The marginal advantage becomes low with more than 200 miners, therefore this number was chosen for the experiments aiming to evaluate the impact of data cachers and the possible improvement brought by the dynamic assignment (D-A) strategy.

DOI: 10.1002/cpe

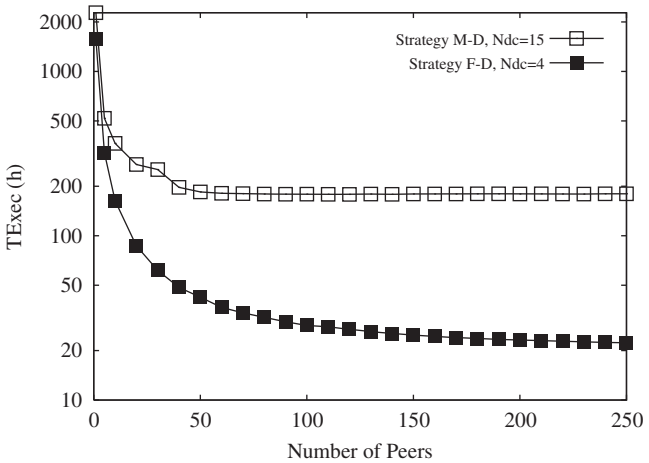


Figure 7. Overall execution time vs the number of active miners. The number of data cachers is set to 4 with strategy M-D and 15 with strategy F-D.

We thus combined the F-D and D-A strategies. A miner, after downloading the entire input data file, executes the assigned job for an amount of time corresponding to a threshold S, which is predetermined. If the job is not finished after this time interval, the miner generates a new job advert that describes the remaining part of the job and sends it to the job manager, for a new assignment. Figure 8 shows the execution time obtained with values of S ranging from 30 s to 57 600 s, or 16 h. Since the largest job executes for 25 300 s, all the values of S larger than this value correspond to disabling the dynamic strategy, because no job needs to be reassigned. The reported results confirm the effectiveness of the D-A strategy. For example, with four data cachers available, the overall execution time can be decreased from 65 700 s (18.25 h) to 52 000 s (14.44 h, in the case that the threshold S is set to 600 s), with a gain of about 21%. The improvement derives from a better load balancing of execution responsibilities among miners: if a job is very large, which could slow down the whole process, the miner can share its load. Notice, however, that the value of the threshold must be chosen with care: if the threshold is too high, the D-A strategy is not fully exploited, whereas, if it is too low, the execution time is lengthened by the large number of job reassignments and the related overhead.

The results discussed so far have been obtained in a scenario in which node failures are very infrequent. In the following, we illustrate the behavior of *Mining@home* in an unreliable environment, where miners may fail ungracefully. We adopted the usual exponential failure distribution model, where the cumulative probability of a node failure increases over time according to the probability function $F(t) = 1 - e^{-\lambda t}$, where λ is the failure rate. The inverse of λ corresponds to the Mean Time Between two successive Failures (MTBF) that occur on a given node. In Figure 9, we studied the behavior of the system on varying the failure rate λ, when adopting the combination of fully download (F-D) and dynamic assignment (D-A) strategy, with a number of four data cachers. Here it is worth recalling that the failure management strategy contemplates that the failure of a

DOI: 10.1002/cpe

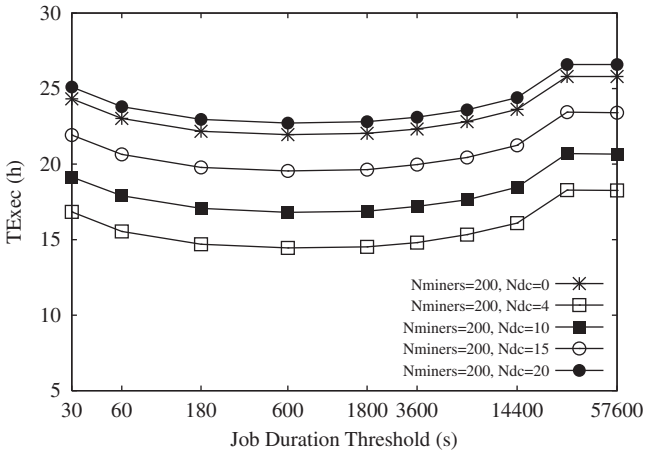


Figure 8. Overall execution time vs the value of the threshold S, when adopting the D-A strategy.

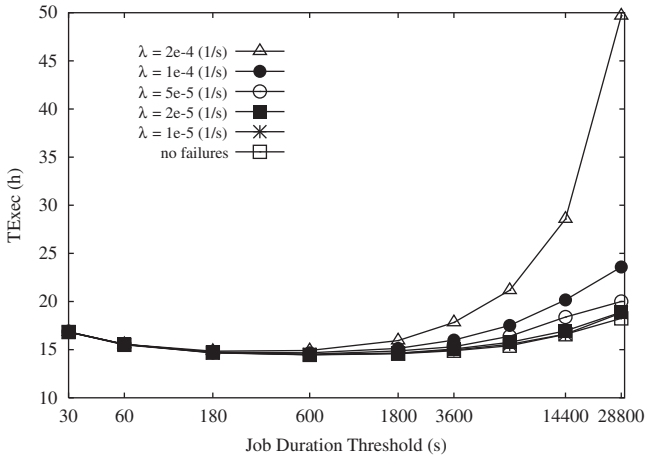


Figure 9. Overall execution time vs the value of the threshold S, when adopting the D-A strategy, for different values of the failure rate λ.

node is detected by the local super-peer, which republishes the job advert of the failed job and allows for its re-assignment.

Figure 9 is interesting in many aspects. First, it shows that the performance of *Mining@home* is hardly affected when the failure rate is low, specifically when its value is not greater than $2 \times 10^{-5}\,\mathrm{s}^{-1}$ or, equivalently, when the MTBF interval is longer than 50 000 s. In the case of higher failure rates, the impact depends on the adopted job duration threshold S. If the threshold is low, jobs are unlikely to fail, as long as S is lower than the MTBF. However, when S is significantly larger than the MTBF, job failures are more frequent, leading to an increase of the overall execution

DOI: 10.1002/cpe

time, as shown in Figure 9 by the trend of the curve that corresponds to a failure rate of $2 \times 10^{-4}\,s^{-1}$, or to an MTBF equal to about 5000 s.

It can be concluded that *Mining@home* can efficiently overcome the failures that occur in an unreliable environment and that the dynamic assignment strategy, which limits the maximum duration of jobs, is useful not only to improve the load balancing characteristics, but also to limit the performance degradation caused by failures.

6. DISCUSSION AND FUTURE WORK

Many directions for future works are open. First, a wide spectrum of improved data dissemination strategies may be exploited to fit the particular problem of closed FIM. In fact, a client may be able to download from data cachers only the rows and columns of the vertical bitmap, i.e. the data set, he does not own already. This would be an improvement w.r.t. the M-D strategy, since the same data set portions are not downloaded again and again, and also w.r.t. the F-D strategy, because useless transactions are never transmitted over the network. We also omitted a set of optimizations, such as compression of transactions and of itemsets collections, which would not affect the general framework we proposed, but, in general, can greatly improve the network efficiency.

Moreover, our data-intensive computing network can be easily extended in the case of multiple data sources, which is very usual in large geographically distributed application. In the future, we plan to tackle other large-scale data mining problems, and test a real implementation of the software on a large number of peers distributed geographically.

7. CONCLUSION

We have proposed *Mining@home*, a novel framework for the exploitation of P2P and/or volunteer computing networks aiming at the execution of large data-intensive data mining tasks.

We illustrated an instantiation of this framework on a particular data mining problem: the extraction of closed frequent itemsets, which is hard to parallelize, and for which a distributed mining algorithm was not proposed so far. Given the peculiarities of the data mining problem, we are confident that our framework can be easily extended to other large data-intensive applications.

Our first contribution was to show that it is possible to decompose the CFIM problem into independent tasks, which however need to share a large volume of data. Our second contribution was to design a novel *data-intensive computing network*, being based on co-operating super-peers with caching capabilities in a P2P topology. We run several simulations of this network, using statistics of a real workload. The results are very promising, in fact, we were able to successfully exploit a large network of 200 peers, and to handle the dissemination of a large volume of data, thus reducing the running time of the mining task from 85 days to about 14 h.

Our approach for distributing large amounts of data across a P2P data mining network opens up a wide spectrum of opportunities. In fact P2P data mining has recently gained a lot of interest. Not only because of the computing power made available by volunteer computing, but also because of new emerging scenarios, such as sensor networks, where data are naturally distributed, and nodes of the network are not reliable. Although many P2P data mining algorithms, such as clustering [32]

DOI: 10.1002/cpe

and feature extraction [33], have been developed, still they suffer the cost of data dissemination. Our approach alleviates this cost, and it can easily deal with failure and load balancing problems. For these reasons, we believe that our proposed data-intensive computing network may be a bridge toward P2P computing for other data mining applications dealing with large amounts of data, such as web documents clustering, or dealing with a naturally distributed environment, e.g. sensor networks.

ACKNOWLEDGEMENTS

This research work has been partially carried out under the Network of Excellences CoreGRID (FP6 contract no. IST-FP6-004265) and S-Cube (FP7 contract no. IST-FP7-215483) funded by the European Commission.

REFERENCES

1. Anderson DP. Boinc: A system for public-resource computing and storage. *GRID '04*: *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing* (*GRID'04*), Pittsburgh, PA, U.S.A., 2004; 4–10.
2. Lucchese C, Orlando S, Perego R. Parallel mining of frequent closed patterns: Harnessing modern computer architectures. *ICDM '07*: *Proceedings of the Fourth IEEE International Conference on Data Mining*, Omaha, NE, U.S.A., 2007.
3. Grahne G, Zhu J. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Transactions on Knowledge and Data Engineering* 2005; **17**(10):1347–1362.
4. Goethals B, Zaki MJ. Advances in frequent itemset mining implementations: Report on FIMI '03. *SIGKDD Explorations Newsletter* 2004; **6**(1):109–117.
5. Buehrer G, Parthasarathy S, Tatikonda S, Kurc T, Saltz J. Toward terabyte pattern mining: an architecture-conscious solution. *PPoPP '07*: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., 2007; 2–12.
6. Agrawal R, Shafer JC. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering* 1996; **8**(6):962–969.
7. Han EH, Karypis G, Kumar V. Scalable parallel data mining for association rules. *SIGMOD '97*: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. ACM: New York, NY, U.S.A., 1997; 277–288.
8. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. *SIGMOD '00*: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, U.S.A., 2000; 1–12.
9. Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: Parallel fp-growth for query recommendation. *RecSys '08*: *Proceedings of the 2008 ACM Conference on Recommender Systems*, Lausanne, Switzerland. ACM: New York, NY, U.S.A., 2008; 107–114.
10. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
11. Anderson DP. Public computing: Reconnecting people to science. *Proceedings of Conference on Shared Knowledge and the Web*, Madrid, Spain, 2003; 17–19.
12. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D. Seti@home: An experiment in public-resource computing. *Communications of the ACM* 2002; **45**(11):56–61.
13. Cappello F, Djilali S, Fedak G, Herault T, Magniette F, Neri V, Lodygensky O. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems* 2005; **21**(3):417–437.
14. Fedak G, Germain C, Neri V, Cappello F. Xtremweb: A generic global computing system. *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, 2001.
15. Moretti C, Steinhaeuser K, Thain D, Chawla NV. Scaling up classifiers to cloud computers. *ICDM '08*: *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, Pisa, Italy. IEEE Computer Society: Washington, DC, U.S.A., 2008; 472–481.
16. Grossman RL, Gu Y, Hanley D, Sabala M, Mambretti J, Szalay A, Thakar A, Kumazoe K, Yuji O, Lee M, Kwon Y, Seok W. Data mining middleware for wide-area high-performance networks. *Future Generation Computer Systems* 2006; **22**(8):940–948.
17. Khoussainov R, Zuo X, Kushmerick N. A toolkit for machine learning on the grid October 2004. *ERCIM News No. 59*.

18. Talia D, Trunfio P, Verta O. Weka4ws: A wsrf-enabled weka toolkit for distributed data mining on grids. *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases* (*PKDD 2005*), Porto, Portugal, 2005.
19. Gu W, Eisenhauer G, Schwan K, Vetter J. Falcon: On-line monitoring for steering parallel programs. *In Ninth International Conference on Parallel and Distributed Computing and Systems* (*PDCS 97*), Washington, DC, U.S.A., 1998; 699–736.
20. Cozza P, Mastroianni C, Talia D, Taylor I. A super-peer protocol for multiple job submission on a grid. *Euro-Par 2006 Workshops* (*Lecture Notes in Computer Science*, vol. 4375), Dresden, Germany. Springer: Berlin, 2007; 116–125.
21. Wille R. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets*, Rival I (ed.). Reidel: Dordrecht, Boston, 1982; 445–470.
22. Yan X, Han J, Afshar R. Clospan: Mining closed sequential patterns in large datasets. *SDM '03*: *Proceedings of the Third SIAM International Conference on Data Mining*, San Francisco, CA, U.S.A., 2003; 166–177.
23. Chi Y, Yang Y, Xia Y, Muntz RR. CMTreeMiner: Mining both closed and maximal frequent subtrees. *PAKDD '04*: *Proceeding of the Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining*, Sydney, Australia, 2004; 63–73.
24. Yan X, Han J. Closegraph: Mining closed frequent graph patterns. *KDD '03*: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, U.S.A., 2003; 286–295.
25. Zaki MJ, Hsiao CJ. Charm: An efficient algorithm for closed itemset mining. *SDM '02*: *Proceedings of the Second SIAM International Conference on Data Mining*, Arlington, VA, U.S.A., 2002.
26. Pei J, Han J, Mao R. Closet: An efficient algorithm for mining frequent closed itemsets. *DMKD '00*: *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Dallas, TX, U.S.A., 2000; 21–30.
27. Lucchese C, Orlando S, Perego R. Fast and memory efficient mining of frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering* 2006; **18**(1):21–36.
28. Cong S, Han J, Padua DA. Parallel mining of closed sequential patterns. *KDD '05*: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, Chicago, IL, U.S.A., 2005; 562–567.
29. Al-Shakarchi E, Cozza P, Harrison A, Mastroianni C, Shields M, Talia D, Taylor I. Distributing workflows over a ubiquitous p2p network. *Scientific Programming* 2007; **15**(4):269–281.
30. Barabási AL, Albert R. Emergence of scaling in random networks. *Science* 1999; **286**(5439):509–512.
31. Lucchese C, Orlando S, Perego R. WebDocs: A real-life huge transactional dataset. *FIMI '04*: *Proceedings of the ICDM 2004 Workshop on Frequent Itemset Mining Implementations*, Brighton, U.K., 2004.
32. Datta S, Bhaduri K, Giannella C, Wolff R, Kargupta H. Distributed data mining in peer-to-peer networks. *IEEE Internet Computing* 2006; **10**(4):18–26.
33. Wurst M, Morik K. Distributed feature extraction in a p2p setting: a case study. *Future Generation Computer Systems* 2007; **23**(1):69–75.

DOI: 10.1002/cpe