Università Ca' Foscari Venezia

Dipartimento di Informatica

Dottorato di Ricerca in Informatica, 2^o ciclo nuova serie

On performance of data mining: from algorithms to management systems for data exploration

Tesi di dottorato di Paolo Palmerini

Coordinatore del dottorato
prof.ssa Simonetta Balsamo

Tutore del dottorato
prof. Salvatore Orlando

to travelers.
those who arrive,
those who do not.

Contents

Acknowledgments

If any value is to be recognized to this work, much of it is due to the endless support that i[1] received from many people. It is my pleasure to thank them all, although here i can only mention a few.

First and foremost, my deep gratitude goes to Salvatore Orlando, for his warm and sharp supervision. I owe him much more than this thesis.

The High Performance Computing Lab. at the ISTI-CNR Institute has been my scientific home for the last five years. Domenico Laforenza, who leads the group, has always been an inspiring scientific reference for my work. Ranieri Baraglia, Renato Ferrini, Raffaele Perego, Fabrizio Silvestri and Giancarlo Bartoli have given me the opportunity to work in a stimulating environment that has guided my research since the very beginning.

My second home was the Computer Science Dept. of the University of Venice. I would like to thank all professors, technical staff and PhD students that have hosted so generously my discontinuous presence. Among them, Moreno Marzolla shared with me this long journey and first introduced me to the immortal "The Next Generation" series.

I spent some six months at the Computer Science Dept. of the Rensselaer Polytechnic Institute where i had the opportunity to join the research group of M. J. Zaki. With him, i thank all the CS Dept. especially Jacky Carley, that welcomed me so friendly. Thanks to all other students in the DMTL group, with whom i shared my icy days in Troy: Joe, Nil, Benjarath, Feng, Saeed, Adnan, Martina, Jeevan and Vinay. A special thank to Bart Goethals, for stimulating and interesting long virtual discussions.

Many friends have courageously kept my name in their address and telephone books and promptly replied to my continuous requests for help: Ferra, Lucia, Kappe, Bia, Francio, Lucia V., Monte, Vanna, Ncussa, Sivia, Bio, Ila, Nicco and Aziz.

Ziggy, Jeff Buckley's voice will always be with me.

During these years, i learned that intellectual traveling is the basis of research. Never settle and rest on one idea, nor on any result. I due this to the Amengià experience, that i am proud to share with Burhan, Sait, Chamuran, Ergian, Musa, Arben, Gasmen, Musli, Usa, Teodora, Marzia, Andrea, Giovanni, Silvia and Francesco.

Without the constant support of my family, this work would have never been finished, nor even started. Uo', Pone, Bobings, Ciccio, Antonietta, Cugginz and my Nonnafranca. Thanks to you all.

Last one: Maddalena, with whom i share the joy of traveling in this life.

[1] i follow E. E. Cummings in not SHOUTING in capital letters when referring to myself.

Abstract

Data Mining (DM) is the science of extracting useful and non-trivial information from the huge amounts of data that is possible to collect in many and diverse fields of science, business and engineering. Due to its relatively recent development, Data Mining still poses many challenges to the research community. New methodologies are needed in order to mine more interesting and specific information from the data, new frameworks are needed to harmonize more effectively all the steps of the mining process, new solutions will have to manage the complex and heterogeneous source of information that is today available for the analysts.

A problem that has always been claimed as one of the most important to address, but has never been solved in general terms, is about the performance of DM systems. Reasons for this concern are: (i) size and distributed nature of input data; (ii) spatio temporal complexity of DM algorithms; (iii) quasi real-time constraints imposed by many applications. When it is not possible to control the performance of DM systems, the actual applicability of DM techniques is compromised.

In this Thesis we focused on the performance of DM algorithms, applications and systems. We faced this problem at different levels. First we considered the algorithmic level. Taking a common DM task, namely Frequent Set Counting (FSC), as a case study, we performed an in depth analysis of performance issues in FSC algorithms. This led us to devise a new algorithm for solving the FSC problem, that we called Direct Count and Intersect (DCI). We also proposed a more general characterization of transactional datasets that allow to forecast, within a reasonable range of confidence, important properties in the FSC process. From preliminary studies, it seems that measuring the entropy of a dataset can provide useful hints on the actual complexity of the mining process on such data.

Performance of DM systems is particularly important for those system that have strong requirements in terms of response time. Web servers are a notable example of such systems: they produce huge amounts of data at different levels (access log, structure, content). If mined effectively and efficiently, the knowledge extracted from these data can be used for service personalization, system improvement or site modification. We illustrate the application of DM techniques to the web domain and propose a DM system for mining web access log data. The system is designed to be tightly coupled with the web server and process the input stream of http requests in an on-line and incremental fashion.

Due to the intrinsically distributed nature of the data being mined and by the commonly claimed need for high performance, parallel and distributed architectures often constitute

the natural platform for data mining applications. We parallelized some DM algorithms, most notably the DCI algorithm. We designed a multilevel parallel implementation of DCI, explicitly targeted at the execution on cluster of SMP nodes, that adopts multi-threading for intra node and message passing for the inter node communications. To face the problems of resource management, we also study the architecture of a scheduler that maps DM applications onto large scale distributed environments, like Girds. We devised a strategy to predict the execution time of a generic DM algorithm and a scheduling policy that effectively takes into account the cost of transferring data across distributed sites.

The specific results obtained in studying single DM kernels or applications can be generalized to wider classes of problems that allows the data miner to abstract from the architectural details of the application (physical interaction with the data source, base algorithm implementation) and concentrate only on the mining process. Following this generic thinking, a Data Mining Template Library (DMTL) for frequent pattern mining was designed at the Rensselaer Polytechnic Institute, Troy (NY) USA, under the supervision of prof. M. J. Zaki. We joined the DMTL project during 2002 fall. We present the main features of DMTL and a preliminary experimental evaluation.

Sommario

Il Data Mining (DM) è la disciplina che si occupa dell'estrazione di informazioni utili e non banali dall'immensa mole di dati che è possibile collezionare in vari campi della scienza e del mondo industriale e commerciale. Il suo relativamente recente sviluppo lascia ancora aperti molti problemi che la comunità scientifica è chiamata ad affrontare e risolvere. Vi è la necessità di nuove metodologie per estrarre informazioni più significative dai dati, nuovi framework si rendono indispensabili per armonizzare tutti i passi del processo di mining, nuove soluzioni sono necessarie per il trattamento di dati complessi ed eterogenei.

Un problema che è sempre stato individuato come di centrale importanza ma mai risolto in termini generali è quello relativo alle prestazioni di sistemi di Data Mining. Le ragioni di questo interesse sono: (i) le dimensioni e la natura distribuita dei dati di input; (ii) la complessità spazio-temporale degli algoritmi di Data Mining; (iii) la richiesta di proprietà quasi di tempo reale per molte applicazioni di DM. Quando non è possibile controllare le prestazioni di sistemi di DM, la reale applicabilità di tali tecniche risulta pregiudicata.

In questa tesi ci concentriamo sulle prestazioni di algoritmi, sistemi e applicazioni di DM.

L'attività di ricerca si è principalmente focalizzata sullo studio delle performance di una tecnica di Data Mining molto utilizzata nelle applicazioni pratiche: l'estrazione di insiemi di item frequenti in database transazionali, o *Frequent Set Counting* (FSC).

Nonostante negli ultimi anni siano stati proposti numerosi algoritmi di FSC che permettono di raggiungere prestazioni sempre migliori, ancora il problema del controllo delle performance di tali algoritmi resta irrisolto. Questo principalmente dovuto al fatto che la complessità della computazione realizzata dipende in modo difficilmente predicibile dalle correlazioni interne al dataset di input e alle sue proprietà statistiche.

Partendo da un'analisi dettagliata delle prestazioni dei migliori algoritmi proposti recentemente, è stato possibile progettare e implementare un nuovo algoritmo di FSC (*Direct Count and Intersect*) (DCI) che in numerosi casi si rivela essere il più performante.

Per poter avere maggiore e completo controllo sulle performance, è inoltre in studio l'analisi di alcune proprietà statistiche dei dataset di input, che permettano di poter predire il costo del FSC su un certo dataset. L'idea di questo studio è quella di caratterizzare la complessità dell'applicazione dell'algoritmo di FSC su un certo dataset, sulla base delle proprietà statistiche del dataset stesso. Alcuni risultati preliminari lasciano dedurre che la misura dell'entropia del dataset sia un buon indicatore dello *sforzo* computazionale richiesto ad un algoritmo di FSC. Sembra quindi possibile poter utilizzare questa misura per la

definizione di strategie di calcolo nell'algoritmo sequenziale (pruning versus compression), di load balancing in quello parallelo e sampling nella versione approssimata dell'algoritmo.

Le performance dei sistemi di Data Mining sono particolarmente importanti in quei sistemi che pongono forti requisiti in termini di tempo di risposta. I server web sono un esempio di tali sistemi: la grande quantità di dati prodotti a vari livelli (log, struttura, contenuto) contiene informazioni potenzialmente utili per personalizzazioni, ottimizzazioni o modificazioni del sito. Viene studiato un sistema di web mining in grado di estrarre informazione sul comportamento degli utenti di un sito dall'analisi degli accessi precedenti e sulla base di questa conoscenza, personalizzare le pagine richieste. Il sistema proposto è realizzato in modo da inserirsi nell'ordinario funzionamento del server web e processare lo stream di richieste httpd in maniera incrementale.

Architetture di calcolo parallele e distribuite costituiscono spesso la piattaforma di calcolo naturale in applicazioni di Data Mining sia per la natura inerentemente distribuita dei dati, sia per la necessità generale di alte prestazioni. In questa tesi viene presentata la parallelizzazione dell'algoritmo DCI, ottimizzata per cluster di SMPs, che adotta un paradigma muti-threading per le comunicazioni interne al nodo e message-passing per quelle tra nodi. Inoltre, viene studiata l'architettura di uno scheduler che mappa applicazioni strutturate di Data Mining su piattaforme distribuite su scala geografica (Knowledge Grid). Viene sviluppata una strategia di scheduling che ottimizza sia il costo computazionale che i costi di trasferimento dei dati.

Un altro filone di ricerca[2] riguarda la realizzazione di sistemi *verticali* di data mining che generalizzino i risultati ottenuti in casi specifici ad una classe più ampia di problemi e permettano di gestire tutte le fasi del processo di data mining, dall'accesso ai dati fino all'estrazione della conoscenza.

Il Data Mining ToolKit (DMTL) è una libreria C++ che permette di sviluppare algoritmi di data mining, rendendo trasparenti all'utente problematiche come l'accesso fisico ai dati o l'implementazione di strutture dati specifiche per un certo problema. L'architettura basata sui template rende inoltre il DMTL flessibile all'estensione verso nuovi pattern e strutture dati.

[2]Questa attività è stata svolta al Rensselaer Polytechnic Institute, sotto la supervisione del prof. M. J. Zaki.

Chapter 1

Introduction

1.1 Why performance matters in Data Mining

The extraction of useful and non-trivial information from the huge amount of data that is possible to collect in many and diverse fields of science, business and engineering, is called Data Mining (DM). DM is part of a bigger framework, referred to as Knowledge Discovery in Databases (KDD), that covers a complex process, from data preparation to knowledge modeling. Within this process, DM techniques and algorithms are the actual tools that analysts have at their disposal to find unknown patterns and correlation in the data.

Typical DM tasks are classification (assign each record of a database to one of a predefined set of classes), clustering (find groups of records that are *close* according to some user defined metrics) or association rules (determine implication rules for a subset of record attributes). A considerable number of algorithms have been developed to perform these and others tasks, from many fields of science, from machine learning to statistics through neural and fuzzy computing. What was a hand tailored set of case specific recipes, about ten years ago, is now recognized as a proper science [81].

It is sufficient to consider the remarkable wide spectrum of applications where DM techniques are currently being applied to understand the ever growing interest from the research community in this discipline. Started as little more than a dry extension of marketing techniques, DM is now bringing important contributions in crucial fields of investigations. Among the traditional sciences we mention astronomy, high energy physics, biology and medicine that have always provided a rich source of applications to data miners [37]. An important field of application for data mining techniques is also the World Wide Web [61]. The Web provides the ability to access one of the largest data repositories, which in most cases still remains to be analyzed and understood. Recently, Data Mining techniques are also being applied to social sciences, home land security and counter terrorism [48] [97].

In this Thesis we will distinguish among a DM algorithm which is a single DM kernel, a DM application, which is a more complex element whose execution in general involves the execution of several DM algorithms, and a DM system, which is the framework within

which DM applications are being executed. A DM system is therefore composed of a software environment that provides all the functionalities to compose DM applications, and a hardware back-end onto which the DM applications are executed.

Due to its relatively recent development, Data Mining still poses many challenges to the research community. New methodologies are needed in order to mine more interesting and specific information from the data, new frameworks are needed to harmonize more effectively all the steps of the mining process, new solutions will have to manage the complex and heterogeneous source of information that is today available for the analysts.

A problem that has always been claimed as one of the most important to address, but has never been solved in general terms, is about the performance of DM systems. Reasons for this concern are:

- The amount of data that it is typically necessary to deal with. *Huge*, *enormous*, *massive* are the terms most commonly used to refer to volumes of data that can easily exceed the terabyte limit;

- Data are generally distributed on a geographical scale. Due to its dimension, but also for other reasons (i.e. privacy, redundancy), data cannot in general be assumed to be placed in a single site. Rather, DM systems have to deal with inherently distributed input sources;

- DM algorithms are often complex both in time and space. Moreover, the complexity of such algorithms depend not only on *external* properties of the input data, like size, number of attributes, number of records, and so on, but also on the data *internal* properties, such as correlations and other statistical features that can only be known at run time;

- Input data change rapidly. In many application domain data to be mined either are produced with high rate or they actually come in streams. In those cases, knowledge has to be mined fast and efficiently in order to be usable and updated.

- DM often supports decision-making process. In this case real-time constraints hold and the request for rapid and efficient results is strong.

When it is not possible to control the performance of DM systems, the actual applicability of DM techniques is compromised. Therefore, if we want to be able to apply such techniques to real problems, involving huge amounts of data in a complex and heterogeneous hardware environment, it is necessary to develop efficient DM algorithms and systems able to support them.

In this Thesis we will focus on the performance of DM algorithms, applications and systems. We will study the efficiency and scalability of single DM kernel but also of more complex DM applications.

1.2 Review of thesis results

In this Thesis we focus on the performance of DM algorithms, applications and systems. We face this problem at different levels. First we consider the algorithmic level (Chapter 2). Taking a common DM task, namely Frequent Set Counting (FSC), as a case study, we perform an in depth analysis of performance issues in FSC algorithms. This led us to devise a new algorithm for solving the FSC problem, that we called Direct Count and Intersect (DCI).

One of the main features of the DCI algorithm is its adaptability to the dataset characteristics in terms of its statistical properties. Starting from this result, we propose a more general characterization of transactional datasets that allow to forecast, within a reasonable range of confidence, some important properties in the FSC process, namely the dataset density, the number of frequent itemsets present in a database and the representativeness of a sample of the entire dataset.

Performance of DM systems is particularly important for those system that have strong requirements in terms of response time. Web servers are a notable example of such systems: they produce huge amounts of data at different levels (access log, structure, content). If mined effectively and efficiently, the knowledge extracted from these data can be used for service personalization, system improvement or site modification. We illustrate (Chapter 3) the application of DM techniques to the web domain and propose a DM system for mining web access log data. The system is designed to be tightly coupled with the web server and process the input stream of http requests in an on-line and incremental fashion.

Due to the intrinsically distributed nature of the data being mined and by the commonly claimed need for high performance, parallel and distributed architectures often constitute the natural platform for data mining applications (Chapter 4). We parallelized some DM algorithms, most notably the DCI algorithm. We designed a multilevel parallel implementation of DCI, explicitly targeted at the execution on cluster of SMP nodes, that adopts multi-threading for intra node and message passing for the inter node communications. To face the problems of resource management, we also study the architecture of a scheduler that maps DM applications onto large scale distributed environments, like Girds [28]. We devised a strategy to predict the execution time of a generic DM algorithm and a scheduling policy that effectively takes into account the cost of transferring data across distributed sites.

The specific results obtained in studying single DM kernels or applications can be generalized to wider classes of problems (Chapter 5) that allows the data miner to abstract from the architectural details of the application (physical interaction with the data source, base algorithm implementation) and concentrate only on the mining process. Following this generic thinking, a Data Mining Template Library (DMTL) for frequent pattern mining was designed at the Rensselaer Polytechnic Institute, Troy (NY) USA, under the supervision of prof. M. J. Zaki. We joined the DMTL project during 2002 fall. We present the main features of DMTL and a preliminary experimental evaluation.

Chapter 2

Frequent Set Mining

Association Rule Mining (ARM) is one of the most popular DM task [3, 26, 31, 81], both for the straight applicability of the knowledge extracted by this kind of analysis and for the wide spectrum of application fields where it can be applied. For example a rule extracted from an hypothetical `tennis.db` database, could be *if weather is cloudy and temperature above 25 degree then don't play*. A rule holds in a database with a *confidence*, which is a percentage measure of how much the rule can be trusted, and a *support*, which is a measure of how many records in the database confirm the rule.

The extraction of association rules from a database is typically composed by two phases. First it is necessary to find the so called *frequent patterns*, i.e. patterns X that occur in a significant number of records. Once such patterns are determined, the actual association rules can be derived in the form of logical implications: $X \Rightarrow Y$ which reads *whenever* X *occurs in a record, most likely also* Y *will occur*.

The computationally intensive part of ARM is the determination of frequent patterns, as confirmed by the considerable amount of efforts devoted to the design of algorithms for this phase of the ARM process.

In this Chapter we review the most significant and recent algorithms for solving the Frequent Pattern Counting problem and present DCI (Direct Count & Intersect), an novel algorithm characterized by significant performance improvements over previous approaches on a wide range of different conditions like input data, parameter values and testbed architecture. In Chapter 4 we describe a parallel and distributed implementation of DCI.

2.1 Problem Statement

As stated above, Association Rule Mining regards the extractions of association rules from a database of transactions, that we will call D. Each transaction in D is a variable length collection of items from a set I. Depending on the actual nature of the database, items can in principle be any kind of object. They can be ISBN codes identifying books in a bookstore, or they can be attribute-value pairs in a relational database, or even more complex data structures like trees or graphs [108]. Since most of the computational complexity of ARM

Symbol	Description
D	Transaction database
I	Set of items appearing in D
k	Length of an itemset, or algorithm step counter
m	Size of I, i.e. number of distinct items in D
m_k	number of distinct items at step k
n	Number of transactions
n_k	Number of transactions at step k
t	A generic transaction
s	Minimum support threshold
F, F_k	Set of frequent itemsets, of length k
C, C_k	Set of candidate itemsets, of length k

Table 2.1: Notation used in this Chapter

is independent from the particular type of data being mined, we consider for our analysis items as integer identifiers.

A rule R has the form $R : X \Rightarrow Y$, where X and Y are sets of items (*itemsets*), such that $(X \cap Y) = \varnothing$. A rule $X \Rightarrow Y$ holds in D with a minimum confidence c and a minimum support s, if at least the $c\%$ of all the transactions containing X also contains Y, and $X \cup Y$ is present in at least the $s\%$ of all the transactions of the database. In order to extract association rules from a database, it is necessary to find all the frequent itemsets in D, i.e. all the set of items whose support overcomes a user defined minimum threshold. This phase is called Frequent Set Counting (FSC).

We review in Table 2.1 the notation used throughout this and the following Chapters.

The database of transaction can be stored in a variety of formats, depending on the nature of the data. It can be a relational database or a flat file, for example. In the rest of this thesis we assume that D is actually a plain file, stored according to some coding conventions but essentially is a plain list of variable length records, without any indexing other than an identifier. We call this a *dataset*. Therefore, a typical dataset could be the one reported on the left of Figure 2.1. This is the so called *horizontal* format, where items are organized in transactions stored sequentially one after the other. On the right of Figure 2.1 we report the so called *vertical* format, where with each item is stored the list of the identifiers of the transactions where the item appears. A third option is to store the dataset in bitmap format, i.e. the dataset is a matrix whose rows are items and columns transaction identifiers. An element in the bitmap is set to 1 if that item appear in the corresponding transaction. We will show later how our algorithm exploits benefits of multiple formats.

The datasets we actually used in our analysis come from a publicly accessible repository of real data, namely the UCI KDD Archive[1]. We also used some synthetic dataset that

[1] `http://kdd.ics.uci.edu`

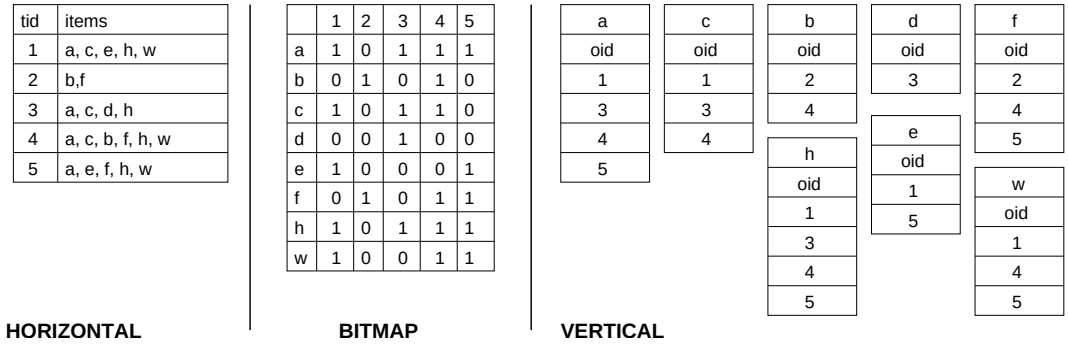


Figure 2.1: A typical transaction dataset in horizontal (left), bitmap (center) and vertical (right) format.

Dataset	n	m	t_{min}	t_{ave}	t_{max}	σ_t	**size (MB)**	**notes**
connect	67557	129	43	43	43	0	11	connect-4 moves
BMS_View_1	59601	497	1	2	267	4.9	1	web clickstreams
chess	3196	75	37	37	37	0	1	chess moves
mushroom	8124	119	23	23	23	0	1	mushrooms features
pumsb	49046	2113	74	74	74	0	14	Census data
pumsb_star	49046	2088	49	50	63	2.0	10	pumsb's less frequent items
USCensus1990	2458285	396	68	68	68	0	665	US Census of 1990
trec	1692082	5267657	1	177	71472	455	1163	Web-TREC corpus
T10I8D400K	338531	1000	1	11	36	4.7	18	synthetic
T25I10D10K	9219	996	10	27	50	5.7	1	synthetic
T25I20D100K	100000	5137	9	28	50	5.7	12	synthetic
T30I16D400K	397487	964	4	29	75	10	50	synthetic

Table 2.2: Characteristics of the datasets used in the analysis. For each dataset we show the number of transactions n, the number of different items m, the minimum, average and maximum transaction length $t_{min}, t_{ave}, t_{max}$, the standard deviation of the transaction length distribution σ_t and the dataset size in MB.

mimic sales data, using the generator developed at IBM[2]. Additionally, we extracted a real-world dataset from the TREC WT10g corpus [11]. The original corpus contained about 1.69 millions of WEB documents. The dataset for our tests was built by considering the set of all the terms contained in each document as a transaction. Before generating the transactional dataset, the collection of documents was filtered by removing HTML tags and the most common words (*stopwords*), and by applying a *stemming* algorithm. The resulting `trec` dataset is huge. It is about 1.1GB, and contains 1.69 millions of short and long transactions, where the maximum length of a transaction is 71,473 items.

We review in Table 2.2 the main features of the datasets used.

[2]http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html

2.2 FSC Algorithms

Despite the considerable amount of algorithms proposed in the last decade for solving the problem of finding frequent patterns in transactional databases (among the many we mention [4] [72] [43] [76] [105] [17] [14] [57]), a single *best* approach still has to be found.

The complexity of the FSC problem relies mainly in the potentially explosive growth of its full search space, whose dimension g is, in the worst case, $g = \sum_{k=1}^{|t_{max}|} \binom{m}{k}$, where t_{max} is the maximum transaction length. If $t_{max} = m$, than the search space is the full power set of I, i.e. $\mathcal{P}(I)$, whose dimension is exponential in m, i.e. $g = 2^m$.

[Apriori]. Taking into account the minimum support threshold, it is possible to reduce the search space, using the *downward closure relation*, which states that an itemset can only be frequent if all its subsets are frequent as well. The exploitation of this property, originally introduced in the *Apriori* algorithm [4], has transformed a potentially exponentially complex problem, into a more tractable one.

Following an iterative level-wise approach, at each iteration k, *Apriori* generates C_k, a set of candidate k-itemsets, and counts the occurrences of these candidates in the transactions. The candidates in C_k for which the minimum support constraint holds are then inserted into F_k, i.e. the set of frequent k-itemsets, and the next iteration is started.

The key concept in the *Apriori* algorithm is the notion of candidate itemset. For example consider a database whose transactions contain the items $\{\{1\},\{2\},\{3\},\{4\}\}$. If we found for some minimum support threshold that 3 is not frequent, i.e. $F_1 = \{\{1\},\{2\},\{4\}\}$, then, when considering itemsets of length $k = 2$ we do not need to check any 2-itemset containing the item 3. Due to the downward closure property, since 3 is not frequent, none of its supersets will be frequent either. So, in this case, the set of candidate itemsets of length 2 would be $C_2 = \{\{1,2\},\{1,4\},\{2,4\}\}$.

Several variations to the original *Apriori* algorithm, as well as many parallel implementations, have been proposed. We can recognize two main methods for determining itemset supports: a *counting*-based approach [4, 40, 72, 14], and an *intersection*-based approach [23, 103]. The former one, also adopted in *Apriori*, exploits a *horizontal* representation of the dataset (left of Figure 2.1), and *counts* how many times each candidate k-itemset occurs in every transaction. If the number of candidates does not explode, this approach permits us to use a limited amount of main memory, since only C_k, along with data structures used to quickly access the candidates (e.g. hash-trees or prefix-trees), are required to be kept in-core. The latter method, on the other hand, exploits a *vertical* dataset representation (right of Figure 2.1), where a *tidlist*, i.e. a list of transaction identifiers (tids), is associated with each itemset, and supports are determined through tidlist intersections. The support of a k-itemset c can thus be computed either by a *k-way* intersection, i.e. by intersecting the k tidlists associated with the k items included in c, or by a *2-way* intersection, i.e. by intersecting the tidlists associated with a pair of frequent $(k-1)$-itemsets whose union is equal to c. While *2-way* methods involve less intersections than

k-way ones, they require much more memory to buffer all the intersections corresponding to (the previously computed) frequent $(k-1)$-itemsets.

[Eclat]. Zaki's *Eclat* algorithm [103] is based on a clever dataset partitioning method that relies on a lattice-theoretic approach for decomposing the search space. Each subproblem obtained from this search space decomposition is concerned with finding all the frequent itemsets that share a common prefix. On the basis of a given common prefix it is possible to determine a partition of the dataset, which will be composed of only those transactions which constitute the support of the prefix. By recursively applying the Eclat's search space decomposition we can thus obtain subproblems which can entirely fit in main memory. Recently Zaki proposed an enhancement to *Eclat* (dEclat) [105], whose main innovation regards a novel vertical data representation called *Diffset*, which only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. Diffsets not only cut down the size of the memory required to store intermediate results, but also increases the performance significantly.

[FP-growth]. Not all FSC algorithms rely on candidate generation [2, 43, 76, 57]. FP-growth [43] builds in memory a compact representation of the dataset, where repeated patterns are represented only once along with the associated repetition counters. The data structure used to store the dataset is called *frequent pattern tree*, or FP-tree for short. The algorithm recursively identifies tree paths which share a common prefix, and projects the tree accordingly, thus reducing its size. These paths are intersected by considering the associated counters. FP-growth works very well for so called *dense* (see Section 2.5) datasets, for which it is able to construct very compressed FP-trees. Note that dense datasets are exactly those for which an explosion in the number of candidates is usually observed. In this case FP-growth is more effective than *Apriori*-like algorithms, which need to generate and store in main memory a huge number of candidates for subset-counting. Unfortunately, FP-growth does not perform well on sparse datasets [109], and this is also confirmed by our tests reported in Section 2.4.6.

[OP]. Recently OP (Opportunistic Projection), another projection-based algorithm, has been proposed [57]. OP overcomes the limits of FP-growth, using a tree-based representation (similar to FP-growth) of projected transactions for dense datasets, and a new array-based representation of the same transactions for sparse datasets. Moreover the authors of OP have proposed a heuristic to opportunely switch between a depth-first and a breadth-first visit of the frequent set tree. The choice of using a breadth-first visit depends on the size of the dataset to mine. For large datasets, the frequent set tree is grown breadth-first, but, when the size of the dataset becomes small enough, the growth of the tree is completed by using a more efficient depth-first approach.

2.3 Open Problems

As demonstrated by direct experiments, no algorithm proposed so far outperforms the others in *all* cases, i.e. on all possible dataset and for all possible interesting values of s. Indeed, a critical source of complexity in the FSC problem resides in the dataset internal correlation and statistical properties, which remain unknown until the mining is completed. Such diversity in the dataset properties is reflected in measurable quantities, like the total number of transactions, or the total number of distinct items m appearing in the database, but also in some other more fuzzy properties which, although commonly recognized as important, still lacks a formal and univocal definition. It is the case, for example, of the notion of how *dense* a dataset is, i.e. how much its transactions tend to resemble among one another.

Several important results have been achieved for specific cases. Dense datasets are effectively mined with compressed data structure [105], explosion in the candidates can be avoided using effective projections of the dataset [57], the support of itemsets in compact datasets can be inferred, without counting, using an equivalence class based partition of the dataset [14].

In order to take advantage of all these, and more specific results, hybrid approaches have been proposed [33]. Critical to this point is *when* and *how* to adopt a given solution instead of another. In lack of a complete theoretical understanding of the FSC problem, the only solution is to adopt an heuristic approach, where theoretical reasoning is supported by direct experience. Therefore, a desirable feature of a new algorithm should be the ability to adapt its behavior to these characteristics and preserve good performances on a wide range of input datasets.

Another important aspect that has strong influence on performance is related to the features of the hardware platform where the algorithm is executed. Modern hw/sw systems need high locality for effectively exploiting memory hierarchies and achieving high performances. Large dynamic data structures with pointers may lack in locality due to unstructured memory references. Other sources of performance limitations may be unpredictable branch instructions. It is important to be able to take advantage of modern operating system optimizations by using simple array data structures, accessed by tight loops which exhibit high spatial and temporal locality.

Much emphasis has been posed in the past in the global reduction of I/O activity in DM algorithms. On the other hand, real datasets are assumed to easily overcome the memory available on any machine. We therefore consider important to adopt an out-of-core approach as a starting point of DM algorithms that will enable us to handle such huge datasets. I/O operations, therefore must be carefully optimized, for example with regular access patterns that allow to achieve high spatial locality and take advantage of prefetching and caching features of modern OSs [12].

When the dataset largely overcomes the physical memory available or poses really strong computational problems, then it is necessary to take advantage of the aggregate resources of parallel and distributed architectures. Efficient parallel implementations of FSC algorithms are therefore necessary in order to be able to efficiently mine huge amounts of data.

2.4 Adaptive Frequent Set Mining: the DCI algorithm

We introduce here the DCI algorithm and illustrate how with its design we face the issues sketched above. Its pseudo-code is reported in Algorithm 1. As *Apriori*, at each iteration DCI builds the set F_k of the frequent k-itemsets on the basis of the set of candidates C_k. However, DCI adopts a hybrid approach to determine the support of the candidates. During the first iterations, it exploits a counting–based technique, and dataset pruning (line 7), i.e. items which will not appear in longer patterns are removed from the dataset. The pruning technique is illustrated in more detail in Section 2.4.3. As soon as the pruned dataset fits into the main memory, DCI starts using an optimized intersection–based technique to access the in-core dataset (line 14 and 16).

Input: D, s
1: $F_1 = \text{first\_scan}(D, s, \&D_2)$; // find frequent items and remove non-frequents from D
2: $F_2 = \text{second\_scan}(D_2, s, \&D_3)$; // find frequent pairs from pruned dataset
3: $k = 2$;
4: // until pruned dataset is bigger than memory...
5: **while** (D_{k+1}.size() > memory_available()) **do**
6: $k++$;
7: $F_k = \text{horizontal\_iter}(D_k, s, k, \&D_{k+1})$; //... keep the horizontal format
8: **end while**
9: // switch to vertical format
10: $dense = D_{k+1}$.is_dense(); // measure dataset density
11: **while** ($F_k \neq \emptyset$) **do**
12: $k++$;
13: **if** ($dense$) **then**
14: $F_k = \text{vertical\_iter\_dense}(D_k, s, k)$; // dense dataset optimization
15: **else**
16: $F_k = \text{vertical\_iter\_sparse}(D_k, s, k, \&D_{k+1})$; // sparse dataset optimization
17: **end if**
18: **end while**

Algorithm 1: Direct Count and Intersect - DCI

As previously stated, DCI adapts its behavior to the features of the specific computing platform and to the features of the dataset mined. DCI deals with dataset peculiarities by dynamically choosing between distinct optimization heuristics according to the dataset density (line 13).

During its initial counting-based phase, DCI exploits an out-of-core, *horizontal* database with variable length records. By exploiting effective pruning techniques inspired by the DHP algorithm [72], DCI trims the transaction database as execution progresses. Dataset pruning entails a reduction in I/O activity as the algorithm progresses, but the main benefits come from the lower computational cost of subset counting due to the reduced number and size of transactions. A pruned dataset D_{k+1} is thus written to disk at each

iteration k, and employed at the next iteration. Let m_k be the number of distinct items included in the pruned dataset D_k, and n_k the number of remaining transactions. Due to dataset pruning $m_{k+1} \leq m_k$ and $n_{k+1} \leq n_k$ always hold. As soon as the pruned dataset becomes small enough to fit into the main memory, DCI adaptively changes its behavior, builds a *vertical* layout database in-core, and starts adopting an intersection-based approach to determine frequent sets. Note, however, that DCI continues to have a level-wise behavior.

At each iteration, DCI generates the candidate set C_k by finding all the pairs of $(k-1)$-itemsets that are included in F_{k-1} and share a common $(k-2)$-prefix. Since F_{k-1} is lexicographically ordered, pairs occur in close positions, and candidate generation is performed with high spatial and temporal locality. Only during the DCI counting-phase, C_k is further pruned by checking whether all the other subsets of a candidate are included in F_{k-1}. Conversely, during the intersection-based phase, since our intersection method is able to quickly determine the support of a candidate itemset, we found much more profitable to avoid this further check. As a consequence, while during its counting-based phase DCI has to maintain C_k in main memory to search candidates and increment their associated counters, this is no longer needed during the intersection-based phase. As soon as a candidate k-itemset is generated, DCI determines its support on-the-fly by intersecting the corresponding tidlists. This is an important improvement over other *Apriori*-like algorithms, which suffer from the possible huge memory requirements due to the explosion of the size of C_k [43].

DCI makes use of a large body of out-of-core techniques, so that it is able to adapt its behavior also to machines with limited main memory. Datasets are read/written in blocks, to take advantage of I/O prefetching and system pipelining [12]. The outputs of the algorithm, e.g. the frequent sets F_k, are written to files that are `mmap`-ped into memory during the next iteration for candidate generation.

In the following we illustrate all the different optimizations and strategies adopted and the corresponding heuristics used to chose among them.

2.4.1 Dynamic type selection

The first optimization is concerned with the amount of memory used to represent itemsets and their counters. Since such structures are extensively accessed during the execution of the algorithm, is it profitable to have such data occupying as little memory as possible. This not only allows to reduce the spatial complexity of the algorithm, but also permits low level processor optimizations to be effective at run time.

During the first scan of the dataset, global properties are collected like the total number of distinct frequent items (m_1) or the maximum transaction size and the support of the most frequent item.

Once this information is available, we remap items to ordered and contiguous integer identifiers. This allows us to decide the best data type to represent such identifiers and their counters. For example if the maximum support with which an item is present is less than 65536, we can use an `unsigned short int` to represent them. The same holds for

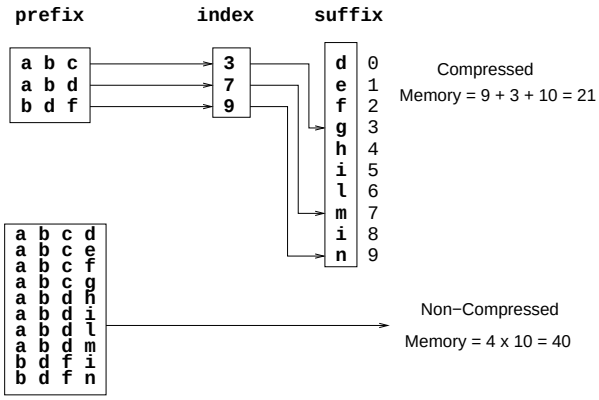


Figure 2.2: Compressed data structure used for itemset collection can reduce the amount of memory needed to store the itemsets.

the remapped identifiers of the items.

The decision of which is the most appropriate type to use for items and counters is taken at run time, by means of a C++ template-based implementation of DCI.

2.4.2 Compressed data structures

Itemsets are often organized in *collections* in many FSC algorithms. Efficient representation of such collections can lead to important performance improvements. In DCI we introduce a compressed representation of an itemset collection, in order to take advantage of prefix sharing among itemsets in the collection, whenever it occurs.

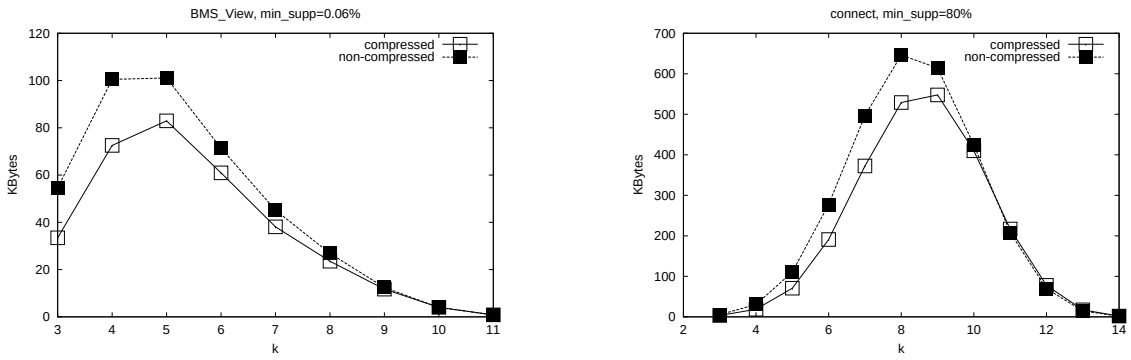
The compressed data structure is based on three arrays (Figure 2.2). At each iteration k, the first array (`prefix`) stores the different prefixes of length $k-1$. In the third array (`suffix`) all the length-1 suffixes are stored. Finally, in the element i of the second array (`index`), we store the position in the `suffix` array of the section of suffixes that share the same prefix. Therefore, when the itemsets in the collection have to be enumerated, we first access the `prefix` array. Then, from the corresponding entry in the `index` array we get the section of suffixes stored in `suffix`, needed to complete the itemsets.

We evaluated the actual reduction in the memory used for two real datasets and reported the results in Figure 2.3.

It is worth noticing that also for the BMS_View_1 dataset, which is known to be sparse, the reduction introduced by the compressed data structures is relevant for most of the iterations.

2.4.3 The counting-based phase

The techniques used in the counting-based phase of DCI are detailed in [67], where we proposed an effective algorithm for mining short patterns. Since the counting-based approach is used only for few iterations, in the following we only sketch the main features of the



(a) (b)

Figure 2.3: Memory usage with compressed itemsets collection representation for BMS_View_1 with $s = 0.06\%$ (a) and connect with $s = 80\%$ (b)

counting method adopted.

In the first iteration, similarly to all FSC algorithms, DCI exploits a vector of counters. For $k \geq 2$, instead of using complex data structures like hash-trees or prefix-trees, DCI uses a novel *Direct Count technique* that can be thought of as a generalization of the technique used for $k = 1$. The technique uses a *prefix table*, PREFIX$_k$[ ], of size $\binom{m_k}{2}$. Each entry of PREFIX$_k$[ ] is associated with a distinct *ordered prefix* of two items. To permit direct access to the different entries of PREFIX$_k$[ ], we used an order preserving, minimal perfect hash function. For $k = 2$, PREFIX$_k$[ ] contains the counters associated with candidate 2-itemsets, while, for $k > 2$, each entry of PREFIX$_k$[ ] stores the pointer to the contiguous sections of ordered candidates in C_k sharing the same prefix. In this case the table is used to count the support of candidates in C_k as follows. For each transaction $t = \{t_1, \ldots, t_{|t|}\}$, we select all the possible 2-prefixes of all k-subsets included in t. We then exploit PREFIX$_k$[ ] to find the sections of C_k which must be visited in order to check set-inclusion of candidates in transaction t.

We designed the out-of-core counting-based phase of DCI having in mind scalability issues. As mentioned above, the dataset is read in blocks, and a pruned dataset D_{k+1} is written to disk at each iteration k. Two different pruning techniques are exploited [67]. The first transforms a generic transaction t, read from D_k, into a pruned transaction $\hat{t}$, which will contain only the items of t that appear in at least $k-1$ frequent itemsets of F_{k-1}. The second technique further prunes $\hat{t}$ during subset counting, and produces the final transaction $\ddot{t}$ which will be written to D_{k+1}. In this case, an item of $\hat{t}$ will be retained in $\ddot{t}$ only if it appears in at least k candidate itemsets of C_k.

Since the counting-based approach becomes less efficient as k increases [84], another important benefit of dataset pruning is that it allows to rapidly reduce dataset size, so that the DCI in-core intersection-based phase can be started early also when huge datasets are mined, or machines with limited memory are used.

2.4.4 The intersection-based phase

DCI starts its intersection-based phase as soon as the vertical representation of the pruned dataset can be stored in core. Hence, at each iteration k, $k \geq 2$, DCI checks memory requirements for the vertical dataset. When the dataset becomes small enough for the computing platform used, the vertical in-core dataset is built on the fly, while the transactions are read and counted against C_k. The intersection-based method thus starts at the following iteration.

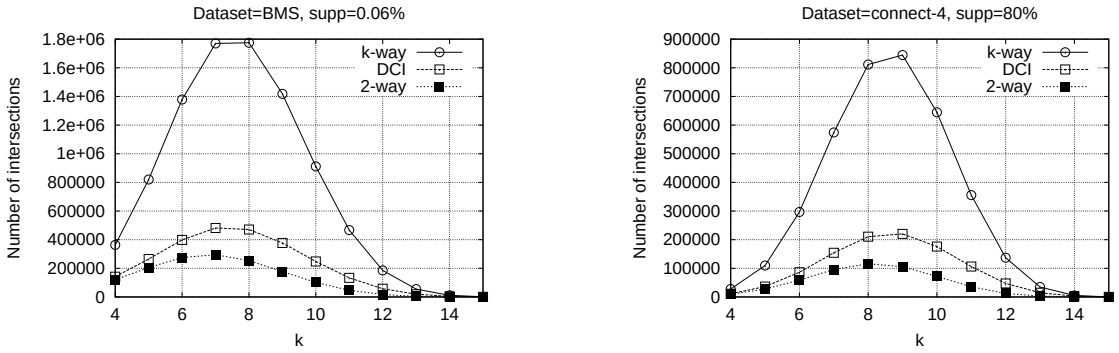
The vertical layout of the dataset is based on fixed length records (tidlists), stored as *bit-vectors* (see Figure 2.9, center). The whole vertical dataset can thus be seen as a bidimensional bit-array whose rows correspond to the bit-vectors associated with non pruned items. Therefore, the amount of memory required to store the vertical dataset is $m_k \times n_k$ bits.

At each iteration of its intersection-based phase, DCI computes F_k as follows. For each candidate k-itemset c, we *and*-intersect the bit-vectors associated with the k items of c (k-*way* intersection), and count the 1's present in the resulting bit-vector. If this number is greater or equal to s, we insert c into F_k. Consider that a bit-vector intersection can be carried out very efficiently and with high spatial locality by using primitive bitwise *and* instructions with word operands. As previously stated, this method does not require C_k to be kept in memory: we can compute the support of each candidate c on-the-fly, as soon as it is generated.

The strategy illustrated above is, in principle, highly inefficient, because it always needs a k-*way* intersection to determine the support of each candidate c. Nevertheless, a caching policy could be exploited in order to save work and speed up our k-*way* intersection method. To this end, DCI uses a small cache buffer to store all the $k-2$ intermediate intersections that have been computed for evaluating the last candidate. Since candidate itemsets are generated in lexicographic order, with high probability two consecutive candidates, e.g. c and c', share a common prefix. Suppose that c and c' share a prefix of length $h \geq 2$. When we process c', we can avoid performing the first $h-1$ intersections since their result can be found in the cache.

To evaluate the effectiveness of our caching policy, we counted the actual number of intersections carried out by DCI on two different datasets: BMS_View_1, and connect. We compared this number with the best and the worst case. The best case corresponds to the adoption of a *2-way* intersection approach, which is only possible if we can fully cache the tidlists associated with all the frequent $(k-1)$-itemsets in F_{k-1}. The worst case regards the adoption of a pure k-*way* intersection method, i.e. a method that does not exploit caching at all. Figure 2.4.(a) plots the results of this analysis on the sparse dataset for support threshold $s = 0.06\%$, while Figure 2.4.(b) regards the dense dataset mined with support threshold $s = 80\%$. In both the cases the caching policy of DCI turns out to be very effective since the actual number of intersections performed is very close to the best case. Moreover, DCI requires orders of magnitude less memory than a pure *2-way* intersection approach, thus better exploiting memory hierarchies.

We have to consider that while caching reduces the number of tidlist intersections, we



(a) (b)

Figure 2.4: Per iteration number of tidlist intersections performed on datasets BMS_View_1 and connect for DCI, and the pure *2-way* and *k-way* intersection-based approaches.

also need to reduce intersection cost. To this end, further heuristics, differentiated w.r.t. *sparse* or *dense* datasets, are adopted by DCI. In order to apply the right optimization, the vertical dataset is tested for checking its density as soon as it is built.

In particular, we compare the bit-vectors associated with the *most frequent items*, i.e., the vectors which are likely to be intersected several times since the associated items occur in many candidates. If large sections of these bit-vectors turn out to be identical, we deduce that the items are highly correlated and that the dataset is dense. In this case we adopt a specific heuristic which exploits similarities between these vectors. Otherwise the technique for sparse datasets is adopted. In the following we illustrate the heuristic in more detail.

2.4.5 Dense vs. Sparse optimization

We adopt two different strategies according to whether a dataset is recognized as *dense* or not. In Section 2.5 describe in more detail the heuristic used to determine a dataset density. Here we illustrate the different strategies adopted once the dataset density is determined:

- **Sparse datasets.** Sparse datasets originate bit-vectors containing long runs of 0's. To speedup computation, while we compute the intersection of the bit-vectors relative to the first two items c_1 and c_2 of a generic candidate itemset $c = \{c_1, c_2, \ldots, c_k\} \in C_k$, we also identify and maintain information about the runs of 0's appearing in the resulting bit-vector stored in cache. The further intersections that are needed to determine the support of c (as well as intersections needed to process other candidates sharing the same 2-item prefix) will skip these runs of 0's, so that only vector segments which may contain 1's are actually intersected. Since information about the runs of 0's is computed once, and the same information is reused many times, this optimization results to be very effective. Note that such technique is a fast approximation of a dataset *projection*, since the vertical dataset is dynamically reduced by

removing transactions (i.e., blocks of columns) that do not support a given 2-item prefix, and thus cannot support larger itemsets sharing the same prefix.

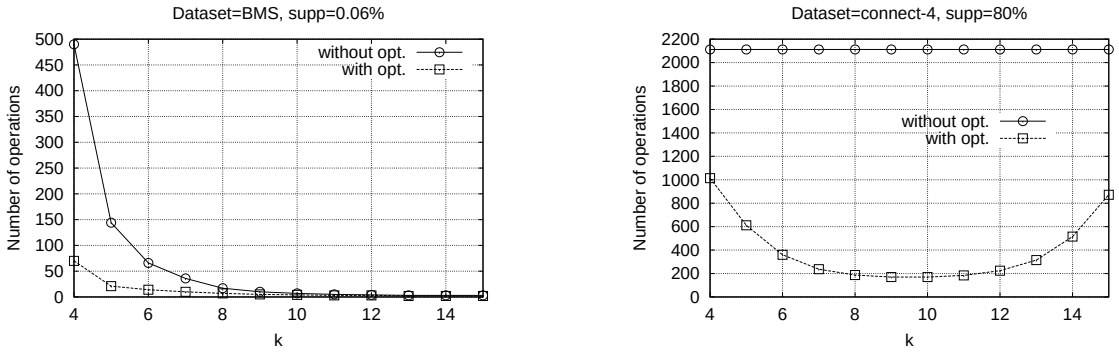
Moreover, sparse datasets offer the possibility of further pruning vertical datasets as computation progresses. The benefits of pruning regard the reduction in the length of the bit-vectors and thus in the cost of intersections. Note that a transaction, i.e. a column of the vertical dataset, can be removed from the vertical dataset when it does not contain any of the itemsets included in F_k. This check can simply be done by *or*-ing the intersection bit-vectors computed for all the frequent k-itemsets. However, we observed that dataset pruning is expensive, since vectors must be compacted at the level of single bits. Hence DCI prunes the dataset only if turns out to be profitable, i.e. if we can obtain a large reduction in the vector length, and the number of vectors to be compacted is small with respect to the cardinality of C_k.

- **Dense datasets.** If the dataset is dense, we expect to deal with strong correlations among the most frequent items. This not only means that the bit-vectors associated with the most frequent items contain long runs of 1's, but also that they turn out to be very similar. The heuristic technique adopted by DCI for dense dataset thus works as follows:

  – reorder the columns of the vertical dataset, in order to move identical segments of the bit-vectors associated with the most frequent items to the first consecutive positions;

  – since each candidate is likely to include several of the most frequent items, we avoid repeated intersections of identical segments. This technique may save a lot of work because (i) the intersection of identical vector segments is done once, (ii) the identical segments are usually very large, and (iii), long candidate itemsets presumably contains several of these most frequent items.

The plots reported in Figure 2.5 show the effectiveness of the heuristic optimizations discussed above in reducing the average number of bitwise *and* operations needed to intersect a pair of bit-vectors. In particular, in Figure 2.5.(a) the *sparse* BMS_View_1 dataset is mined with support threshold $s = 0.06\%$, while in Figure 2.5.(b) the *dense* dataset connect is mined with support threshold $s = 80\%$. In both cases, we plotted the per-iteration cost of each bit-vector intersection in terms of bitwise *and* operations when either our heuristic optimizations are adopted or not. The two plots show that our optimizations for both sparse and dense datasets have the effect of reducing the intersection cost up to an order of magnitude. Note that when no optimizations are employed, the curves exactly plot the bit-vector length (in words). Finally, from the plot reported in Figure 2.5.(a), we can also note the effect of the pruning technique used on sparse datasets. Pruning has the effect of reducing the length of the bit-vectors as execution progresses. On the other hand, when datasets are dense, the vertical dataset is not pruned, so that the length of bit-vectors remains the same for all the intersection-based iterations.



(a) (b)

Figure 2.5: Average number of bitwise *and* operations actually performed to intersect two bit-vectors as a function of k.

2.4.6 Experimental Results

For the sequential tests we used a MS-Windows2000 workstation equipped with an AMD Athlon XP 2400 processor, 1GB of RAM memory and an eide hard disk.

The performance of DCI were compared with those achieved under the same testing conditions by three of the most efficient FSC algorithms proposed in literature. In particular, we used FP-growth, OP, and dEclat, in the original implementation by the respective authors[3].

Figure 2.6 reports the total execution times obtained running FP-growth, dEclat, OP, and DCI on various datasets as a function of the support threshold s.

We can see that DCI performances are very similar to those obtained by OP. In some cases DCI slightly outperforms OP, in other cases the opposite holds. In all the tests conducted, DCI instead outperforms both dEclat and FP-growth. Finally, due to its adaptiveness, DCI can effectively mine huge datasets like USCensus1990 and TREC, on which all the other algorithms fail also using very large support thresholds (Figure 2.7).

To validate DCI design choices, we evaluated its scale-up behavior when varying both the size of the dataset and the size of available memory. The dataset employed for these tests was synthetically generated (T20I8M1K) with variable number of transactions. The support threshold was set to $s = 0.5\%$, while the available RAM in a Pentium II 350MHz PC was changed from 64MB to 512MB by physically plugging additional memory banks into the PC main board. Figure 2.8.(a) and 2.8.(b) plot several curves representing the execution times of DCI and FP-growth, respectively, as a function of the number of transactions contained in the dataset processed. Each curve plotted refers to a series of tests conducted with the same PC equipped with a different amount of memory. As it can be seen from Figure 2.8.(a), DCI scales linearly also on machines with less memory. Due to its adaptiveness and the use of efficient out-of-core techniques, it is able to modify its

[3]We acknowledge J. Han, M. J. Zaki and J. Liu for kindly providing us the latest version of their codes.

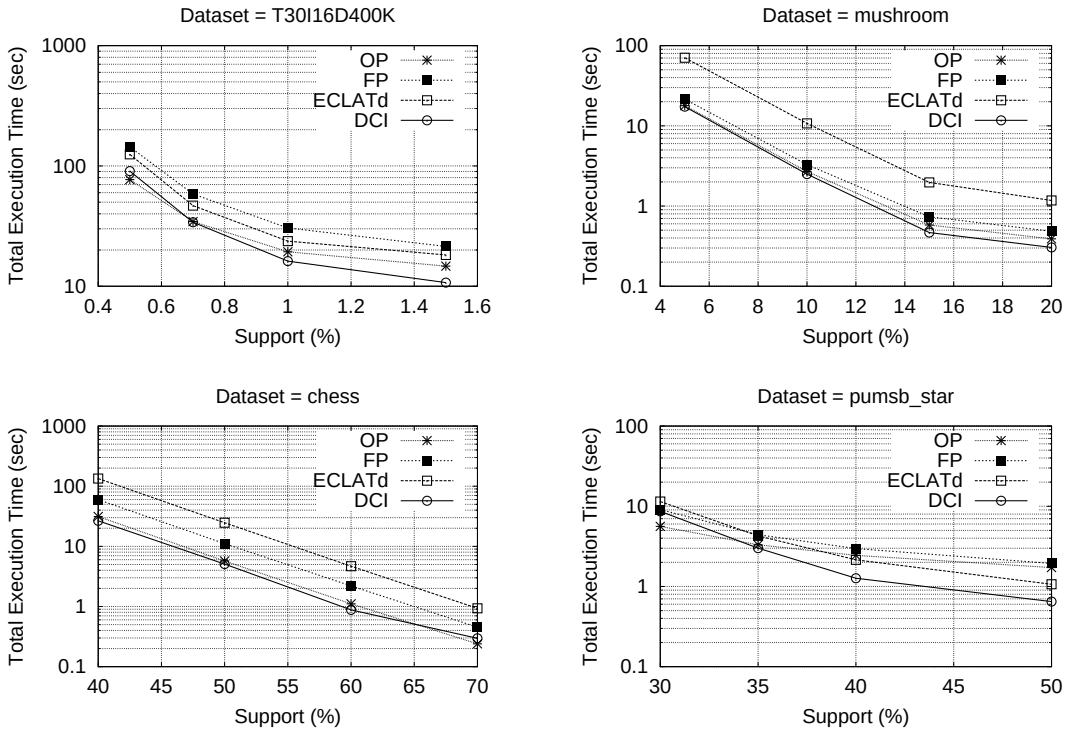


Figure 2.6: Total execution times of OP, FP-growth, dEclat, and DCI on various datasets as a function of the support threshold.

behavior in function of the features of the dataset mined and the computational resources available. For example, in the tests conducted with two millions of transactions, the in-core intersection-based phase was started at the sixth iteration when only 64MB of RAM were available, and at the third iteration when the available memory was 512MB. On the other hand the results reported in Figure 2.8.(b) show that FP-growth requires much more memory than DCI, and is not able to adapt itself to memory availability. For example, in the tests conducted with 64MB of RAM, FP-growth requires less than 30 seconds to mine the dataset with 200K transactions, but when we double the size of the dataset to 400K transactions, FP-growth execution time becomes 1303 seconds, more than 40 times higher, due to a heavy page swapping activity.

2.5 Characterization of dataset properties

As already stated in this Chapter, one important property of transactional datasets, is their *density*. The notion of density, although not yet formally defined in the literature, plays a crucial role in determining the best strategy for solving the FSC problem. A dataset is said to be dense, when most transactions tend to be similar among them: they have about the

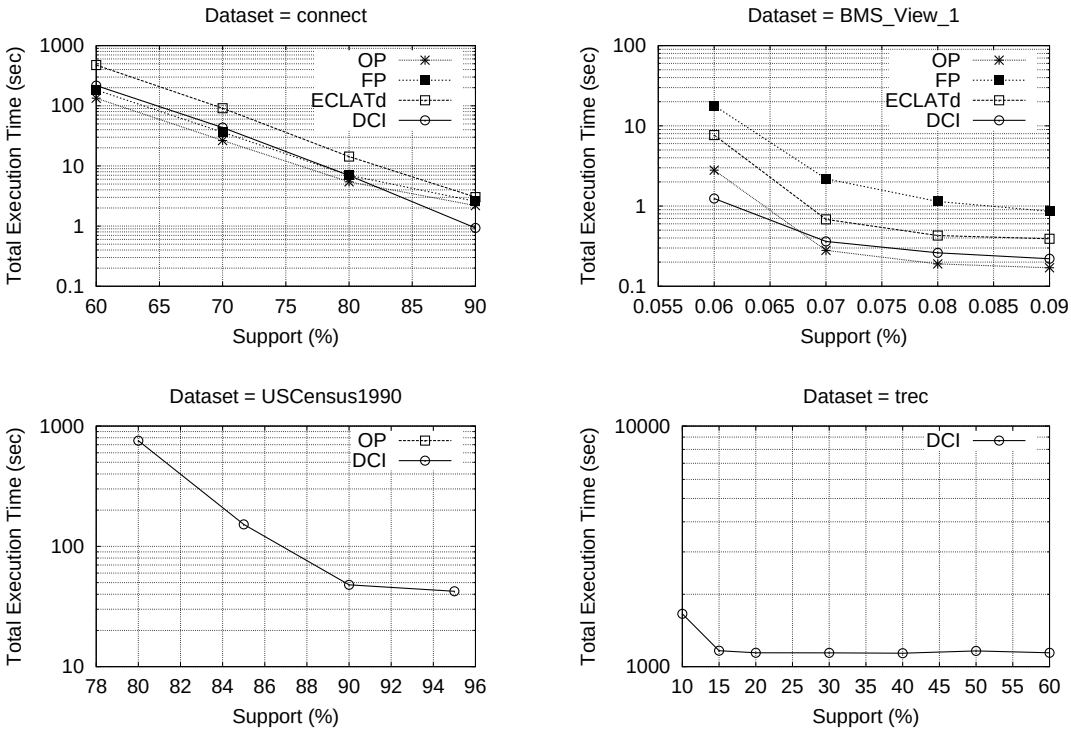
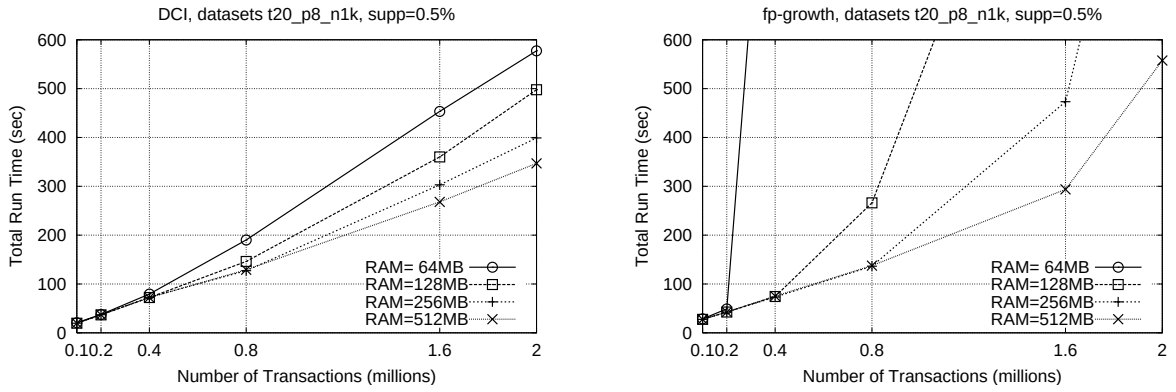


Figure 2.7: Total execution times of OP, FP-growth, dEclat, and DCI on various datasets as a function of the support threshold.

same length and contain mostly the same items. When dealing with such datasets, several optimizations can be very effective, like using compressed data structures both for the dataset and for itemsets representation. Conversely, for sparse datasets where transactions differ a lot one from another, in general it might be useful to apply some pruning technique in order to get rid of useless items and transactions. We showed in Section 2.4.5 the details of such optimizations.

Another important issue is to determine the range of support threshold within which a *relevant* number of frequent itemsets will be found. When nothing is known about the dataset, the only option is to start mining with a high support value and then decrease the threshold until we find a number of frequent itemsets that fulfill our requirements. It would be nice if we could have an idea of the dataset behavior for all possible supports, before starting the actual - potentially expensive - computation.

In a recent work [32] Goethals *et al.* have analytically found a tight upper bound for the number of candidates that can be generated during the steps of a level-wise algorithm for FSC. From the knowledge of the number of frequent patterns found at step k, it is possible to know an upper bound for the candidates that will be generated at step $k+1$. This permits to estimate with a good level of accuracy the maximal pattern length, i.e. how many steps will be performed by the algorithm. Such knowledge is used by the authors



(a) (b)

Figure 2.8: Total execution times of (a) DCI, and (b) FP-growth, on dataset T20I8M1K with variable number of transactions mined with a support threshold $s = 0.5\%$ on a PC equipped with different RAM sizes as a function of the number of transactions (ranging from $100K$ to $2M$).

to postpone the actual counting of candidates as much as possible, thereby limiting the number of database scans, without the risk of a combinatorial explosion in the number of candidates.

Rather than focusing on the characteristics of level-wise FSC algorithms, namely candidate generation, the problem we want to address in this Section is that of finding a good characterization of a dataset complexity, in terms of the number of patterns satisfying a given support threshold, and of its density, in order to apply adequate optimization strategies.

In [82], Zaki *et al.* study the length distribution of frequent and maximal frequent itemsets for synthetic and real datasets. This characterization allows them to devise a procedure for the generation of benchmarking datasets that reproduce the length distribution of frequent patterns.

To the best of our knowledge, no previous work has been addressed at a characterization of a dataset density and at how this characterization can be used for optimization purposes.

We will try to answer the questions of whether a dataset is dense or sparse and which is the support range of potential interest for a given dataset [68].

2.5.1 Definitions

We define here the notion of dataset density, i.e. of how much the transactions inside the dataset resemble one with another. Intuitively, the more dense a dataset is, the more its transactions will differ only for very few items.

The following two limit cases give an idea of the intuitive meaning of density. The maximum density that a dataset can have corresponds to all the transactions being identical. On the other hand, minimum density corresponds to each transaction containing only one

single item and each item appearing in only one transaction. In Figure 2.9 we represent the dataset in binary format, where to each of the n transactions we associate a row in a matrix whose elements are 0 or 1 according to whether the corresponding item - ranging from 1 to m - is present or not in the transaction. We see that maximum density - Fig. 2.9 (a) - corresponds to a matrix whose elements are all 1, and minimum density - Fig. 2.9 (b) - to the identity matrix, with ones only along the main diagonal. In the following we will respectively refer to these two extreme datasets as $\mathcal{D}$ (for dense) and $\mathcal{S}$ (for sparse).

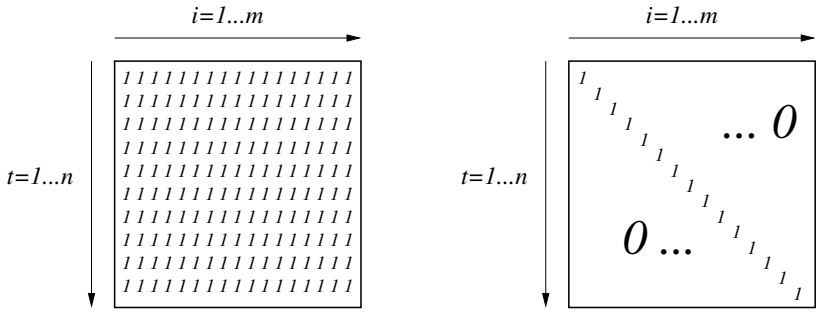


Figure 2.9: Limit cases of maximum (a) and minimum (b) density for transactional datasets in binary format.

Real datasets of course in general exhibits an intermediate behavior with respect to the limit cases. In Figure 2.10 we plotted the bitmap representation[4] of some real datasets and a synthetic one.

From this simple representation it is already possible to isolate some qualitative feature of a dataset. It is for example evident that BMS_View_1 and the synthetic dataset (Figure 2.10 (d) and (e) respectively) are more sparse than the others. Nevertheless, while the synthetic dataset shows a regular structure, with an almost uniform distribution of ones in the bitmap, BMS_View_1 exhibits a more complicated internal structure, with few extremely long transactions and a few items appearing in almost all transactions (almost full rows and columns respectively).

2.5.2 Heuristics

We would like to be able to characterize datasets with a measurable quantity from which it is possible to tell how dense a dataset is, i.e. if it is closer to $\mathcal{D}$ or to $\mathcal{S}$.

The simplest choice we can make, is to consider the fraction of 1's in the dataset matrix, which is 1 for $\mathcal{D}$ and $1/m$ for $\mathcal{S}$. Such definition is not sufficient to capture all the interesting dataset features from the point of view of FSC, since two transactions with the

[4]The bitmap is obtained by evaluating the number of occurrences of each distinct item in a subset of n/m transactions and assigning a level of gray proportional to such count.

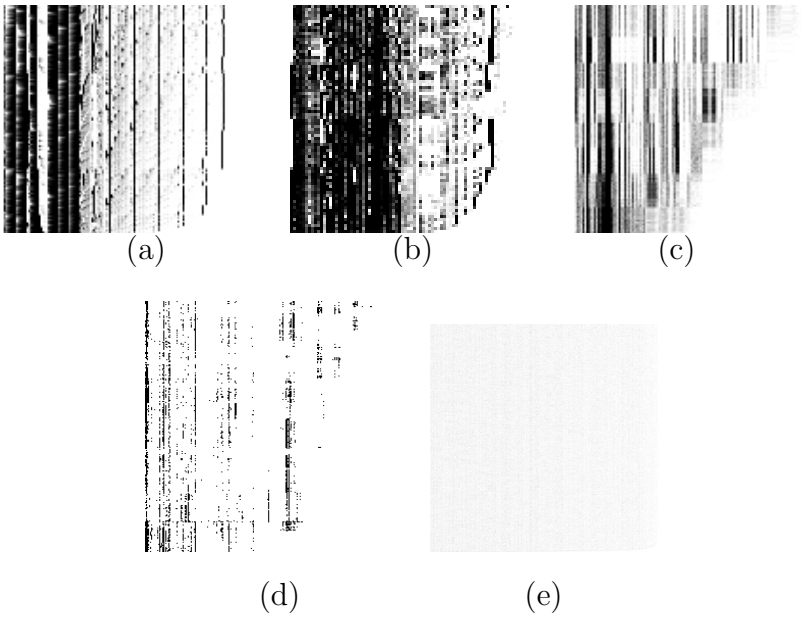


Figure 2.10: Bitmap representation of datasets: (a) connect; (b) chess; (c) mushroom; (d) BMS_View_1; (e) T25I10D10K. All images have been rescaled to the same squared size.

same number of items will bring the same contribution to the overall density, regardless of how they actually resemble one another.

A more interesting behavior is exhibited by the average support of frequent items, plotted in Figure 2.11.

For almost all values of the support threshold - along the x axis - dense datasets maintain an average support of frequent items that is sharply higher than the threshold, i.e. curves corresponding to dense datasets reside well above the $y = x$ line. On the other hand, for sparse dataset only a few items have support high enough to pass the threshold filter, even for low values of the threshold. We can use this qualitative analysis two define two classes of datasets. Dense datasets are characterized by an average support for frequent items that is *much* higher than the minimum support threshold for almost all supports. How *much* higher is a question that remains unsolved at this level of analysis.

Although we can qualitatively classify a dataset into sparse/dense categories using the average support of frequent items, we still have the same problem stated above: transaction similarity is not taken into account and therefore we have little or no chance at all to have any hints on the frequent patterns contained in the dataset.

We can have a more accurate measure of transaction similarity by measuring the correlation among the tidlists corresponding to the most frequent itemsets. This is actually the heuristic adopted in the DCI algorithm for determining whether a dataset is dense or not. In the following, we give a description of such heuristic.

We basic idea is to measure the area in the bitmap vertical representation of the dataset where tidlists are almost the same. If such area exists and is wide enough, then the dataset

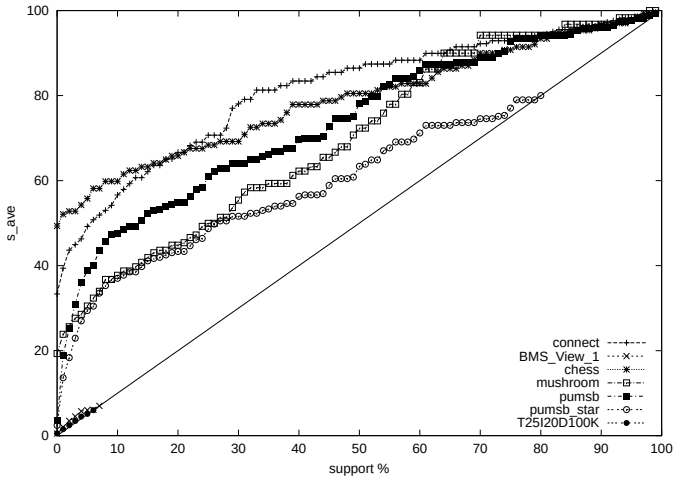


Figure 2.11: Average support

is dense.

As an example, consider the two dataset in Figure 2.12, where tidlists are placed horizontally, i.e. rows correspond to items and columns to transactions. Suppose to choose a density threshold $\delta = 0.4$. If we order the items according to their support, we have the most frequent items, i.e. those who are more likely to appear in long itemsets, at the bottom of each figure. Starting from the bottom, we find the maximum number of items whose tidlists have a significant common section. In the case of dataset (a), for example, a fraction $f = 1/2$ of the items share $p = 90\%$ of the transactions, i.e. 90% of the times they appear, they are together. So we have $d = fp = 0.5 \times 0.9 = 0.45$ which is above the density threshold. For dataset (b) on the other hand, a smaller section $p = 50\%$ is common

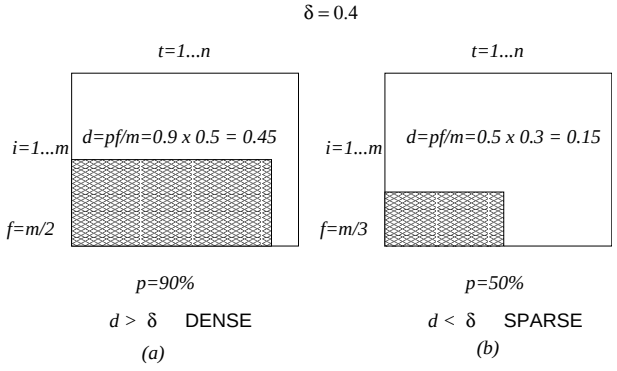


Figure 2.12: Heuristic to establish a dataset density or sparsity

to $f = 1/3$ of the items. In this last case the density $d = fp = 0.3 \times 0.5 = 0.15$ is lower than the threshold and the dataset is considered as sparse. It is worth to notice that since this notion of density depends on the minimum support threshold, the same dataset can exhibits different behaviors when mined with different support thresholds.

2.5.3 Dataset Entropy

To have a more accurate estimate of transaction similarity, we should measure the amount of information contained in each transaction. The information relevant to the FSC problem is the collection of itemsets contained in a transaction. Our idea is to consider the dataset as a transaction source and measure the entropy of the signal - i.e. the transactions - produced by such source.

For a given length k, we define the following quantity:

$$H_k(s) = -\sum_{i=1}^{\binom{m}{k}} [\![p_i > s]\!]\, p_i \log p_i \tag{2.1}$$

where p_i is the probability of observing itemset i, of length k, in the dataset and the truth function $[\![expr]\!]$ which equals 1 if $expr$ is TRUE and 0 otherwise, is used to select only the frequent itemsets.

The probabilities p_i are the normalized frequencies:

$$p_i = \frac{f_i}{\sum_j f_j} \tag{2.2}$$

with f_i being equal to the number of transaction where the itemset appears, divided by the number n of transactions in the dataset.

The intuitive idea behind Equation 2.1 is that of considering the dataset as source *emitting* a signal, i.e. transactions. The information contained in such source are the itemsets. By selecting only frequent itemsets, we consider the minimum support threshold influence on the problem complexity: the lower the minimum support, the harder the mining process.

Definition 2.1 holds for any itemset length k. Measuring $H_k(s)$ for all possible k corresponds to running an FSC algorithm. We therefore ask if it is possible to capture useful information on the dataset properties only considering small values for k.

We begin considering H_1, i.e. the entropy of single items, which is the simplest H_k to be evaluated. We consider the two limit cases $\mathcal{S}$ and $\mathcal{D}$ with no minimum support filter applied. In the first dataset $f_i = 1/m, \forall i$ so $p_i = 1/m, \forall i$ and $H_1(\mathcal{S}) = \log(m)$. For the dense dataset, $f_i = 1, \forall i$ so $p_i = 1/m \forall i$ and again $H_1(\mathcal{D}) = \log(m)$. Therefore using H_1 we are not able to distinguish between the two limit cases which are interesting for our purposes. This is due to the fact that with H_1 we cannot differentiate between the contribution of a transaction of, say, n_t items and n_t transactions each one with only one of the items of the first transaction. For example a dataset composed by one transaction

$D = \{\{1,2,3\}\}$ and a dataset composed by three transactions $D' = \{\{1\},\{2\},\{3\}\}$ would have the same H_1.

The problem is that in H_1 we are not considering any correlation among items, i.e. we have no notion of transaction. The simplest level of correlation we can consider is that of single items correlations. We therefore go one step further and consider $k = 2$:

$$H_2(s) = -\sum_{i=1}^{m(m-1)/2} [\![p_i > s]\!]\, p_i \log p_i \tag{2.3}$$

Now for the sparse dataset $\mathcal{S}$ we have $f_i = p_i = 0, \forall i$, since the itemsets we consider are the $m(m-1)/2$ pairs of m items. So $H_2(\mathcal{S}) = 0$. On the other hand, for $\mathcal{D}$ we have $f_i = 1, p_i = 2/(m(m-1))\forall i$ and $H_2(\mathcal{D}) = \log(m(m-1)/2)$.

Using H_2 it is possible to distinguish between $\mathcal{D}$ and $\mathcal{S}$. Of course also H_2 fails to fully characterize a dataset. The following two datasets: $D = \{\{1,2,3,4\}\}$ and $D' = \{\{1,2\},\{3,4\}\}$ are again indistinguishable looking only at the pair occurrences, as in H_2, and H_3 should be considered instead.

In Figure 2.13 we plotted the measured value of $H_2(s)$ for different supports and different datasets.

It is possible to identify two different qualitative behaviors. A subset of all the datasets considered, while increasing s, only a little variation in H_2 is observed. For another subset of the datasets considered this is not true. This second group exhibits a rapid decay of H while increasing s.

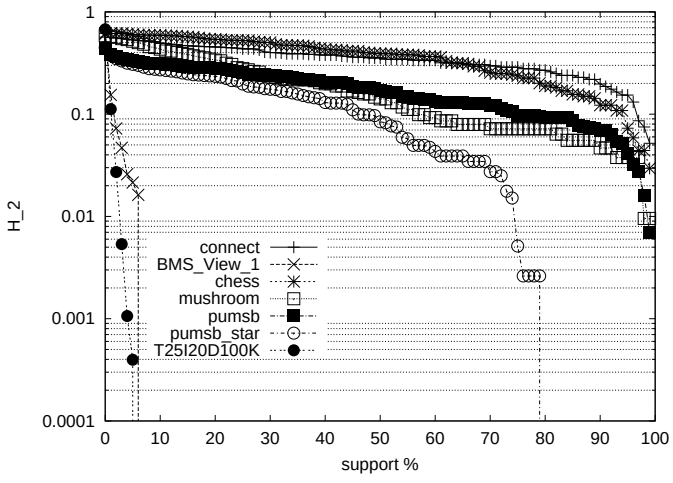


Figure 2.13: Entropy of known datasets. Sparse dataset are characterized by a rapidly decreasing entropy, while dense datasets maintain an almost constant value for all supports.

Dataset	$s_{min} \sim s_{max}$ (%)
connect	$45 \sim 90$
chess	$15 \sim 70$
mushroom	$1 \sim 20$
pumsb	$60 \sim 95$
pumsb_star	$25 \sim 60$
T25I20D100K	$0.55 \sim 1$
BMS_View_1	$0.06 \sim 0.4$

Table 2.3: Support range for each dataset in Figure 2.14. Ten support values were considered within each range.

It is important to notice that the variation of H_2 is significant more than its absolute value. In other words in order to compare the characteristics of two datasets, it is necessary to evaluate $H_2(s)$ for several supports.

We conclude this section with an observation on the computational cost of such measure. The evaluation of H_1 only involves single items frequencies, therefore a single dataset scan is required while the amount of memory is of $O(m)$. To evaluate H_2 it is necessary to know the pair frequencies, which implies a further dataset scan or, if enough memory is available - $O(m^2)$ - everything can be evaluated in the first scan.

2.5.4 Entropy and Frequent itemsets

One interesting feature of H is that it can be related with the total number of frequent itemsets present in a dataset for a given support threshold.

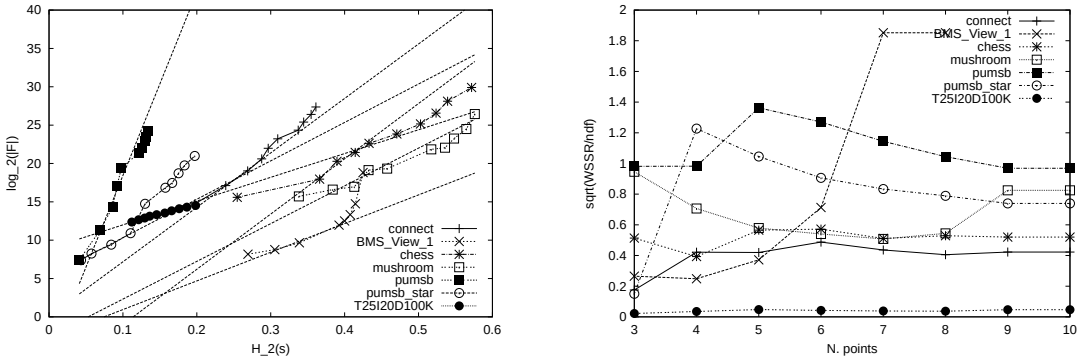
When applying an FSC algorithm to an unknown dataset, one option is to start with a very high support threshold and then keeps running the FSC algorithm while lowering the threshold, until a satisfactory number of frequent patterns is found. This approach has the strong limitation that we do not know in advance how long can the computation last, even for high minimum supports, and, most importantly, it is not possible to determine how many patterns the FSC algorithm will find for a given support until we run it.

Since H is related to the correlation among transactions, one could think that its variation can be related to the variation in the number of patterns found. In fact, this conjecture is confirmed by experimental verification.

We consider the total number of frequent itemsets found for different datasets and supports, and measure the corresponding value of $H_2(s)$. For each dataset we considered 10 different support values in different ranges, reported in Table 2.3.

We found that the logarithm of the total number of frequent patterns is linearly correlated with $H_2(s)$. In Fig. 2.14 (a) a linear fit obtained by a minimum square regression, is superimposed to each data series.

Therefore, if we wanted to know how many frequent patterns are contained in a dataset for a given support threshold, we could run the FSC algorithm with a few values of high



(a) (b)

Figure 2.14: Total number of frequent patterns found versus entropy and the error on predicting the number of frequent itemsets produced

supports (corresponding to small execution times), and then extrapolate the unknown number of frequent patterns for the requested support. Conversely, from the same linear regression, we could determine which is the support that will produce a given number of frequent patterns.

In Figure 2.14 (b) we evaluated the accuracy of the estimation obtained from the linear regression. For each dataset, we performed a linear fit taking a variable number of points from the curves in Fig. 2.14 (a), starting from the highest supports, i.e lowest values for $H(s)$. Then we plotted the χ^2 of the fit, taking an increasing number of points in the fit. The lower the value of χ, the better the quality of the fit.

From this results we can assert that it is possible to estimate the number of total frequent itemsets present in a dataset for a given support, by extrapolating the values obtained for high supports.

In Figure 2.15 we plot the average error on the estimate of the number of frequent itemsets using an increasing number of points from the plot in Figure 2.14 (a).

For each dataset, we performed a linear fit taking a variable number of points from the curves in Fig. 2.14 (a), starting from the highest supports, i.e lowest values for $H(s)$. Then we evaluated the average error obtained when estimating the number of frequent patterns for the rest of the points in the plot. More precisely, if we have n different values of $H(s)$ (in our case $n = 10$), and the n corresponding values for the total number of frequent patterns $|F|$, we perform the linear fit for j ($j \geq 3$) of such $(H, \log(|F|))$ pairs. From the fit parameters we can estimate the values of the remaining $n - j - 1$ number of frequent patterns $\overline{|F|}$. The error of the estimate, as a function of j, is defined as:

$$error(j) = \frac{1}{n-j-1} \sum_{i=j+1}^{n-1} \frac{|F|_i - \overline{|F|}_i}{|F|_i} \cdot 100\% \tag{2.4}$$

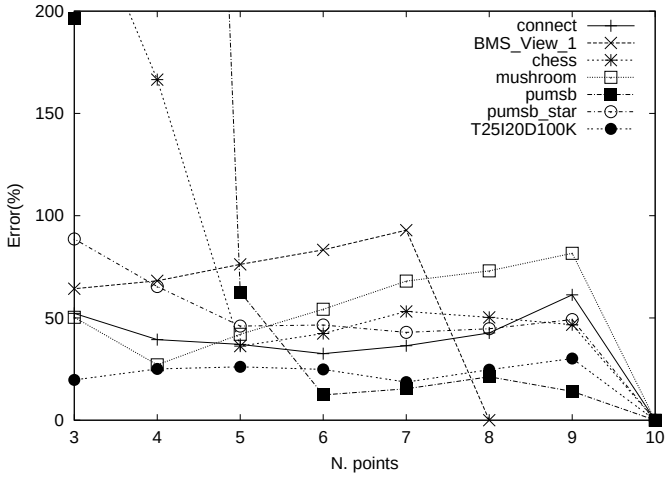


Figure 2.15: A measure of the error obtained on the estimate of the total number of candidates

From Figure 2.15 we can see that for most datasets already for four points the errors are lower than 40%, which permits to have a reasonable confidence when predicting the total number of frequent itemsets. In particular, by running the FSC algorithm with four values of s - that can be chosen in order to minimize the execution time - we are able to predict the total number of frequent patterns found for any value of s, within a 40% confidence.

2.5.5 Entropy and sampling

A common problem that arises in datamining when dealing with huge amount of data is that of obtaining an approximate result by applying an algorithm on a sample of the input dataset. In this case it is of interest to determine how accurate is the knowledge extracted from the sample. Several sampling algorithms have been proposed with variable level of accuracy [21] [107], yet the problem remains of determining whether a sample is a good representative of the entire dataset or not, without running the mining algorithm.

We argue that a sample whose entropy, as given by Eq. 2.1, is similar to the entropy of the entire dataset, will more likely produce the same amount of frequent itemsets. We show how the results from a set of experiments on synthetic and real datasets, confirm this conjecture.

We define:

$$\Delta O_s = \sum_{k}^{k_{max}} \frac{(|F_k| - |F_k^s|)}{\sum_k^{k_{max}} |F_k|} * 100\% \tag{2.5}$$

where k_{max} is the maximum between the maximal length of frequent itemsets in the sample and in the real dataset, F_k is the set of frequent itemsets of length k in the entire dataset and F_r^s is the same in the sample.

In terms of metrics 2.5, a good sample will have $\Delta O_s = 0$. In a series of tests, we show how the quality of a sample can be actually correlated with a variation in the entropy. In Figure 2.16 we plotted ΔO versus the relative variation of entropy ΔH, for 200 random samples of six datasets. A linear fit is superimposed to the experimental data, showing the effective correlation between the two quantities.

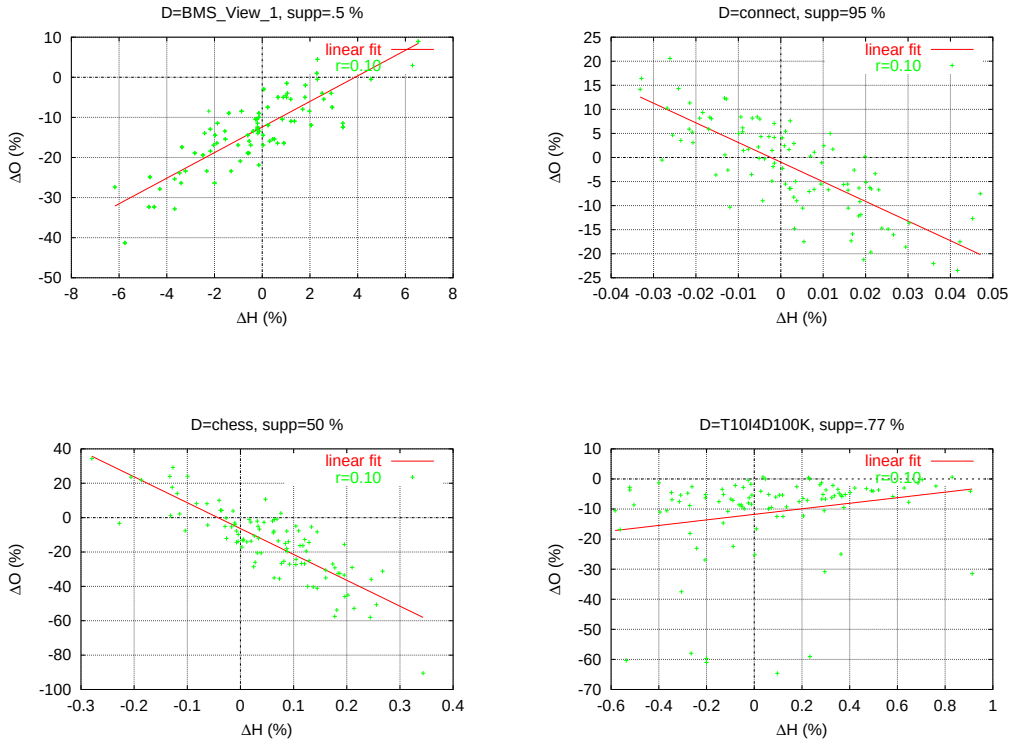


Figure 2.16: Correlation between variation in the output and variation in the entropy. Each plot refer to a different dataset. Points inside a plot refer to different random samples of the same dataset.

This result suggests that the entropy of a dataset is a good measure of the relevant statistical properties of a dataset. Even without running the complete FSC algorithm, we

can use entropy to measure how representative a sample is. An interesting problem would be of finding an entropy preserving sampling algorithm.

Chapter 3

Web mining

The Web is a rich source of information. An exponentially increasing set of multimedia documents, a huge amount of computational and storage resources, used to manage such documents, the related access patterns by a correspondingly increasing set of users. Extracting useful knowledge from this apparently endless mine of information, is a challenge that has been faced at many levels.

One of the possible application of the so called *Web Mining*, i.e. DM on web data, is to analyze data relative to user navigation sessions on a web server. Such sub-domain of Web DM is called Web Usage Mining and is aimed at developing personalization or recommendation systems. For such systems to be effective, the concern on performance impose the utilization of efficient techniques, able to extract and use valuable knowledge during the web server normal functioning with a limited overhead over its standard operations. Web Usage Mining (WUM) systems are typically composed by two parts. An offline analysis of the server access logs, where suitable categories are built, and then an online classification of active requests, according to the offline analysis.

In this Chapter we propose a recommendation system, implemented as a module of the Apache web server, that is able to dynamically generate suggestions to pages that have not yet been visited by users and might be of potential interest. Differently from previously proposed WUM systems, SUGGEST incrementally builds and maintain the historical information, without the need for an offline component, by means of an incremental graph partitioning algorithm. We analyze the quality of the suggestions generated and the performance of the module.

3.1 Problem description

Web Mining is a popular technique that can be used to discover information "hidden" into Web-related data. In particular, Web Usage Mining (WUM) is the process of extracting knowledge from web users access or clikstream, by exploiting Data Mining (DM) technologies. It can be used for different goals such as *personalization*, *system improvement* and *site modification*.

Typically, the WUM personalization process is structured according to two components, performed online and offline with respect to the web server activity [63], [101], [64], and [13]. The offline component is aimed at building the base knowledge used in the online phase, by analyzing historical data, generally the server access log files. The main functions carried out by this component are *Preprocessing*, i.e. data cleaning and session identification, and *Pattern Discovery*, i.e. the application of DM techniques, like association rules, clustering or classification.

The online component is devoted to the generation of personalized content. On the basis of the knowledge extracted in the offline component, it processes a request to the web server by adding personalized content which can be expressed in several forms such as links to pages, advertisements, information relating to products or service estimated to be of interest for the current user.

3.2 Web Usage Mining Systems

In the past, several WUM projects have been proposed to foresee users' preference and their navigation behavior, as well as many recent results improved separately the quality of the personalization or the user profiling phase [66] [65] [30]. We focus on the architectural issues of WUM systems, by proposing an incremental personalization procedure, tightly coupled with the web server ordinary functionalities.

We review in the following the most significant WUM projects, which can be compared with our system.

Analog [101] is one of the first WUM systems. It is structured according to two main components, performed online and offline with respect to the web server activity. Past users activity recorded in server log files is processed to form clusters of user sessions. Then the online component builds active user sessions which are then classified into one of the clusters found by the offline component. The classification allows to identify pages related to the ones in the active session and to return the requested page with a list of related documents. The geometrical approach used for clustering is affected by several limitations, related to scalability and to the effectiveness of the results found. Nevertheless, the architectural solution introduced was maintained in several other more recent projects.

In [63] B. Mobasher *et al.* presents WebPersonalizer a system which provides dynamic recommendations as a list of hypertext links to users. The analysis is based on anonymous usage data combined with the structure formed by the hyperlinks of the site. In order to obtain aggregate usage profiles DM techniques such as clustering, association rules, and sequential pattern discovery are used in the preprocessing phase. Using such techniques, server logs are converted in session clusters (i.e. sequence of visited page) and pageview clusters (i.e. set of page with common usage characteristics). The online phase considers the active user session in order to find matches among the users activities and the discovered usage profiles. Matching entries are then used to compute a set of recommendations which will be inserted into the last requested page as a set of a list of hypertext links. Recently the same author proposed [65] a hybrid personalization model that can dynamically chose

between either non-sequential models (like association rules or clustering) or models where the notion of time ordering is maintained (like sequences). The decision of which model to used is based on an analysis of the site connectivity. WebPersonalizer is a good example of the two level architecture for personalization systems.

Our system SUGGEST inherits most of its concepts from PageGather [77] a personalization system concerned with the creation of index pages containing topic-focused links. The site organization and presentation is enhanced by using automatically extracted users' access patterns. The system takes as input a Web server log and a conceptual description of each page at a site and uses a clustering methodology to discover pages that are visited together. To perform this task a *co-occurrence* matrix M is built where each element M_{ij} is defined as the conditional probability that page i is visited, given that page j has been visited in the same session. A threshold minimum value for M_{ij} allows to prune some uninteresting entries. The directed acyclic graph G associated with M is then partitioned finding the graph's cliques. Finally, cliques are merged to originate the clusters. This solution introduced the hypotheses that users behave coherently during their navigation, i.e. pages within the same session are in general conceptually related. This assumption is called *visit coherence*. The generated static index pages are kept in a separate "Suggestion Section" of the site. The main limitation of this approach is the loosely coupled integration of the WUM system with the web server ordinary activity. The system we propose is implemented as a module of the Apache web server. This architectural choice allows us to deliver personalized content with limited impact on the web server performance.

In [100] SpeedTracer, a usage mining and analysis tool is described. Its goal is to understand the surfing behavior of users. Also in this case the analysis is done by exploring the server logs entries. The main characteristic of SpeedTracer is that it does not require cookies or user registration for session identification. In fact, it uses five kind of information: IP, Timestamp, URL of the requested page, Referral, and Agent to identify user sessions. The application uses innovative inference algorithms to reconstruct user traversal paths and identify user sessions. Advanced mining algorithms uncover users' movement through a Web site. The final result is a collection of valuable browsing patterns which help webmasters better understand user behavior. SpeedTracer generates three types of statistics: user-based, path-based and group-based. User-based statistics pinpoint reference counts by user and durations of access. Path-based statistics identify frequent traversal paths in Web presentations. Group-based statistics provide information on groups of Web site pages most frequently visited.

As we have observed in the introduction, in most of the previous works a two-tiers architecture is generally used. The use of two components leads to the disadvantage to have an "asynchronous cooperation" between the components themselves. The offline component has to be periodically performed to have up-to-date data patterns, but how frequently is a problem that has to be solved on a case-specific basis.

On the other hand the integration of the offline and online component functionalities in a single component, poses other problems in terms of the overall system performance, which should have a negligible impact on user response times. The system must be able to generate personalized content in a fraction of user session. Moreover, the knowledge mined

by the single component has to be comparable, if not better, with that mined by the two separate components.

We focus our attention on the architecture of WUM systems, and not on the algorithms used for the personalization process. As an extension of a previous work [13] we propose a WUM system whose main contribution with respect to other solutions is to be composed of only one component that is able to update incrementally and automatically the knowledge obtained from historical data and online generate personalized content.

3.3 Web personalization with SUGGEST

3.3.1 Architecture

SUGGEST is implemented as a module of the Apache [94] web server, to allow an easy deployment on potentially any kind of web site currently up and running, without any modification of the site itself. The knowledge model is updated as requests arrive at the server in an incremental fashion and used directly on the new requests in order to produce personalized content.

Schematically, SUGGEST works as follows. As http requests arrive at the server: (i) user navigation sessions $\{A\}$ are built, (ii) the underlying knowledge base is updated in the form of page clusters $\{L\}$, (iii) active user sessions are classified in one of the clusters, (iv) finally a list of suggestions $\{S\}$ is generated and appended to the requested page u. Collapsing the two tiers into a single module pushed aside the asynchronous cooperation problem: i.e. the need to estimate the update frequency of the knowledge base structure.

In Algorithm 2 the steps carried out by SUGGEST are presented. At each step SUGGEST identifies the URL u requested (line 1) and the session to which the user belongs (line 2). With the session id it retrieves the identifier of the URL v from which the user is coming (line 3). According to the current session characteristics it updates the knowledge base and generate the suggestions to be presented to the user. All this steps are based on a graph-theoretic model which represents the aggregate information about the navigational sessions.

3.3.2 The Session Model

To extract information about navigational patterns, our algorithm models the usage information as a complete graph $G = (V, E)$. The set V of vertexes contains the identifiers of the different pages hosted on the web server. The set of edges E is weighted using the following relation:

$$W_{ij} = \frac{N_{ij}}{max\{N_i, N_j\}} \tag{3.1}$$

where N_{ij} is the number of sessions containing both pages i and j, N_i and N_j are respectively the number of sessions containing only page i or page j. The rationale of Equation 3.1 is to measure of how many times pair of pages are visited together, but discriminating

internal pages from the so called *index pages*. Index pages are those which, generally, do not contain useful contents and are used only as a starting point for a browsing session. For this reason dividing N_{ij} by the maximum occurrence of the two pages reduces the relative importance of links involving index pages. Index pages are very likely to be visited with any other page and nevertheless are of little interest as potential suggestions. The data structure we used to store the weights is an adjacency matrix M where each entry M_{ij} contains the value W_{ij} computed according to formula 3.1.

Input: $M, \{L\}, \{A\}, u$
Output: A list $\{S\}$ of suggestions

1: $page\_id_u$ = Identify_Page(u); // Retrieves the id of the URL u by accessing a trie built on top of all of the existing URLs.
2: $session\_id$ = Identify_Session(); // Using cookies.
3: $page\_id_v$ = Last_Page($session\_id$); // Returns the last page visited during the current session.
4: PW = A[$session\_id$];
5: **if** (!Exists($page\_id_u$, $page\_id_v$, PW)) **then**
6: // pages (u,v) are not present in an active session
7: $M[page\_id_u, page\_id_v]$++;
8: **if** (($W_{uv} > MinFreq$) $\wedge$ ($L[u] \neq L[v]$)) **then**
9: MergeCluster(L[u], L[v]); // Merges the two clusters containing u and v.
10: **end if**
11: **end if**
12: **if** (!Exists($page\_id_u$, PW)) **then**
13: $M[page\_id_u, page\_id_u]$++;
14: New_L = Cluster(M,L, $page\_id_u$);
15: $L = New\_L$;
16: **end if**
17: push(u, PW);
18: S =Create_Suggestions($session\_id$,L);
19: return(S);

Algorithm 2: The SUGGEST algorithm. Inputs are the cooccurence matrix M, the list of clusters L, the active user session A, the user request u.

Before computing the weights in M, it is first necessary to identify user sessions. One option would be to apply some heuristics based on the IP address and time-stamp. The main drawbacks of this approach are due to the presence of users behind proxies of NATs[1]. In this case, in fact, those users appear as a single one coming from the NAT (or gateway) machine. In SUGGEST users' sessions are identified by means of cookies stored on the client side. Cookies contain the *keys* to identify the clients' sessions. Once a key has been retrieved by our module, it is used to access a table which contains the corresponding session id. The computational cost of such operation is thus relatively small. In fact, a

[1]Network Address Translators.

hash table can be used to store and retrieve the keys allowing this phase to be carried out in time $O(1)$.

3.3.3 Clustering Algorithm

SUGGEST finds groups of strongly correlated pages by partitioning the graph according to its connected component. The algorithm performed by this phase of SUGGEST is shown in Algorithm 3. SUGGEST actually uses a modified version of the well known incremental connected components algorithm [25]. Given an initial partitioning of the graph G into a set of clusters $\{L\}$ and a request page u, we need to determine if and how such request modified the cluster partitioning of the graph. We start from u a Depth First Search (DFS) on the graph induced by M and we search for the connected component reachable from u. Once the component is found, we check if there are any nodes in previus component not considered in the visit. If we find any such node, than the previously connected component has been split and we continue applying the DFS until all the nodes have been visited. In the worst case (when all the site URLs are in the same cluster and there is only one connected component) the cost of this algorithm will be quadratic in the number of pages of the site (to be more precise it will be linear in the number of edges of the complete graph G). Actually, to reduce the contributions of poorly represented link, the incremental computation of the connected components is driven by two threshold parameters. Aim of these thresholds is to limit the number of edges to visit by:

1. filtering those W_{ij} below a constant value. We called this value $minfreq$. Links (i.e. elements M_{ij} of M) whose values are less than $minfreq$ are poorly correlated and thus not considered by the connected components algorithm;

2. considering only components of size greater than a fixed number of nodes, namely $MinClusterSize$. All the components having size lower than a threshold value are discarded because considered not significant enough.

In general, the incremental connected component problem can be solved using an algorithm working in $O(|V| + |E|\mathcal{A})$ time, where $\mathcal{A} = \alpha(|E|, |V|)$ is the inverse of the Ackermann's function[2]. This is the case in which we have the entire graph and we would incrementally compute the connected component by adding one edge at a time. Our case is slightly different. In fact, we do not deal only with edge addition but also with with edge deletion operations. Moreover, depending on the value chosen for *minfreq*, the number of clusters and their sizes will vary, inducing a variation in the number of edges considered in the clusters restructuring phase.

3.3.4 Suggestion generation

After this last step, we have to construct the actual suggestions list (Algorithm 4). This is built in a straightforward manner by finding the cluster which have the largest intersection

[2]The Ackermann's functions grows very fast so that its inverse grows very slowly.

Input: *session_id*, $page\_id_u$, $A[session\_id]$, $L : L[page\_id_u] = cluster\_id$.
Output: An updated clustering structure.

```
1: ret_val = L; clust=L[u];
2: C = {n ∈ [1..|L|] | L[n] = clust};
3: h = pop(C);
4: ret_val[h] = h;
5: clust = h;
6: F ≠ ∅;
7: while h ≠ NULL do
8:    for all (i ∈ G s.t. W_hi > MinFreq) do
9:       remove(C,i);
10:      push(F,i);
11:      ret_val[i]=clust;
12:   end for
13:   if F = ∅ then
14:      pop(F,h);
15:   else
16:      if (C ≠ ∅) then
17:         pop(C, h);
18:         clust = h;
19:      else
20:         h = NULL;
21:      end if
22:   end if
23: end while
24: return(ret_val);
```

Algorithm 3: The clustering phase.

with the current session. The final suggestions are composed by the most relevant pages in the cluster, according to the order determined by the clustering phase. The cost of this algorithm is proportional to the size of active sessions $\{A\}$ and thus is constant ($O(1)$).

3.3.5 Experimental Evaluation

We performed an experimental evaluation of the SUGGEST module performance by measuring both its effectiveness and its efficiency. The first measure is the most difficult to define because it implies to be able to measure how much a suggestion has been useful for the user. In [13] we introduced a quantity that allows to quantify the effectiveness - i.e. the quality - of the suggestions. Such measure was based on the intersection of real sessions with the corresponding set of suggestions. For every session S_i composed by n_i pages there is a set of suggestions R_i, generated by the module in response to the requests

Input: $session\_id$, $page\_id_u$, $A[session\_id]$, $L : L[page\_id_u] = cluster\_id$.
Output: A list of url identifiers, i.e. the suggestions $\{S\}$.

clust= 0; max_rank= 0; ret_val= $\emptyset$;
for $(i = 0; i < |A[session\_id]|; i++)$ **do**
  rank[i] = $|\{n \in A[session\_id] \mid L[n] = L[A[session\_id][i]]\}| + 1$;
  if (rank[i] > max_rank) **then**
    max_rank = rank[i];
    clust=$L[A[session\_id][i]]$;
  end if
end for
$C = \{n \in L \mid L[n] = \text{clust}\}$;
for all $(c \in C)$ **do**
  for $(i = 0; i < NUMSUGGESTIONS; i++)$ **do**
    if $((c \in A[session\_id] \vee (W_{cu} < W_{u,\text{ret\_val}[i]}))$ **then**
      break;
    else
      shift(ret_val, i);
      ret_val[i] = c;
    end if
  end for
end for

Algorithm 4: The suggestions building phase

in S_i. The intersection between S_i and R_i is:

$$\omega_i^{old} = \frac{|\{p \in S_i | \, p \in R_i\}|}{n_i} \tag{3.2}$$

With this measure we are not able to capture the potential impact of the suggestions on the user navigation session. For example, if a page that the user would visit at the end of the session is instead suggested at the beginning of the session, the suggestion in this case could help the user finding a shorter way to what she/he is looking for. Therefore we extend expression 3.2 taking into account the distance of the suggestions generated with the actual pages visited during the session.

For every user session S_i, we split the session into two halves. The first half S_i^1 is used to generate a set of suggestions R_i^1, the second half is used to measure the intersection with the suggestions. For every page p_k that belongs to the intersection $S_i^2 \cap R_i^1$ and appears in position k within S_i^2, we add a weight $f(k)$. We choose f so that more importance is given to pages actually visited at the end of the session. The simplest choice is to take

$$f(k) = k \tag{3.3}$$

A different form for f could have been chosen. For example to have the same coverage measure as used in [65] it is sufficient to take $f(k) = 1$, or any constant value. It is also

possible to increase the importance of the pages non linearly by taking $f(k) = k^2$.

In conclusion, for the whole session log, the measure of the quality of the suggestions is given by

$$\Omega = \sum_{i=1}^{N_S} \frac{\sum_{k=1}^{n_i/2} [\![p_k \in \{S_i^2 \cap R_i^1\}]\!] \frac{f(k)}{F}}{N_S} \tag{3.4}$$

where N_S is the number of sessions and $[\![expr]\!]$ is the truth function equal to 1 if expr is TRUE, 0 otherwise. F is simply a normalization factor on the weights, i.e. $F = \sum_{j=1}^{n_i/2} f(j)$.

Starting from real life access log files, we generated requests to an Apache server running SUGGEST and recorded the suggestions generated for every navigation session contained in the log files. We considered three real life access log files, publicly available[3]: NASA, USASK and BERKLEY containing the requests made to three different web servers. We report in Table 3.1 the main features of such datasets.

Dataset	**Time window**	N_s
NASA	27 days	19K
USASK	180 days	10K
BERK	22 days	22K

Table 3.1: Access log files used to measure the quality of suggestions.

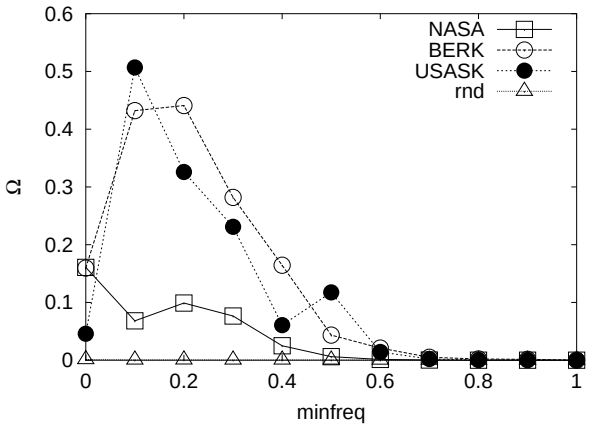
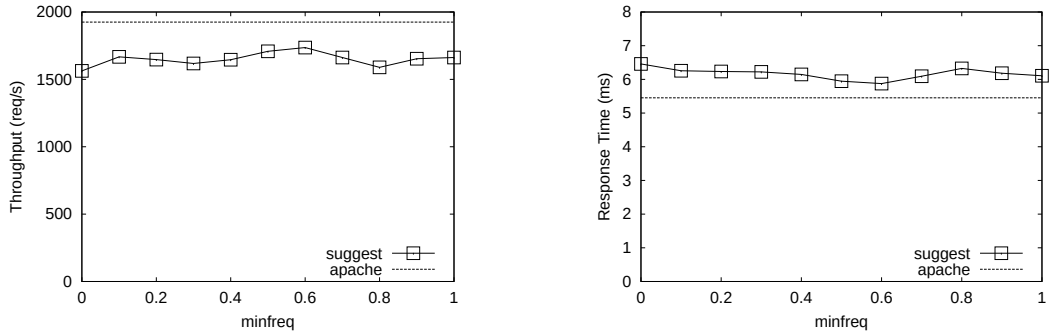


Figure 3.1: Coverage of suggestions for the NASA, BERK, USASK access log files, varying *minfreq*. We also plotted the result of a random suggestions generator.

[3] `www.web-caching.com`

For each dataset we measured Ω. The results obtained are reported in Figure 3.1. In all curves, varying the *minfreq* parameter, we measure Ω for the suggestions generated by SUGGEST. We also plotted the curve relative to the suggestions generated by a random suggestion generator. As it should be expected, the random generator performs poorly and the intersection between a random suggestion and a real session is almost null. On the other hand, suggestions generated by SUGGEST show a higher quality Ω, which, in all dataset, reaches a maximum around *minfreq*=0.2.



(a) (b)

Figure 3.2: Impact of SUGGEST on the throughput of Apache (a) and Apache response time with and without the SUGGEST module (b)

In order to measure the impact of the SUGGEST module on the overall performance of the http server, we plotted the throughput (Figure 3.2 (a)), i.e. number of http requests served per time unit, and the total time needed to serve one single request (Figure 3.2 (b)).

As we can see from the Figures, the impact of SUGGEST on the Apache web server is relatively limited, both in terms of throughput and in terms of the time need to serve a single requests. This limited impact on performance makes SUGGEST suitable to be adopted in real life production servers. It must be noticed, however, that in its current version SUGGEST allocates a buffer of memory whose dimension is quadratic in the number of pages in the site. This might be a severe limitation in sites whose number of pages can be of many thousands.

Chapter 4

Distributed Data Mining

Data Mining technology is not only composed by efficient and effective algorithms, executed as stand alone kernels. Rather, it is constituted by complex applications articulated in the non trivial interaction among hardware and software components, running on large scale distributed environments. This last feature turns out to be both the cause and the effect of the inherently distributed nature of data, on one side, and, on the other side, of the spatio-temporal complexity that characterizes many DM applications. For a growing number of application fields, Distributed Data Mining (DDM) is therefore a critical technology.

In this Chapter, after reviewing the state of the art and open problems in DDM (Section 4.1), we describe the parallelization of DCI (Section 4.2), the Frequent Set Counting algorithm introduced in Chapter 2. We developed a parallel and distributed version of DCI, targeted at the execution on an SMP-based cluster of workstations. In order to be able to optimize resource usage in a multi programmed large scale distributed platform, we study (Section 4.3) the issues related to the scheduling of DM jobs on Grid [28] environments.

4.1 State of the art and open problems

4.1.1 Motivation for Distributed Data Mining

Due to the logistic organization of the entities that collects data – either private companies or public institutions – data are often distributed at the origin. The sales point of a large chain as Wal-Mart, or the branches of a bank or the census offices in a country, these are all examples of data collected in a distributed fashion. Such data are typically too big to be gathered at a single site or, for privacy issues, can only be moved, if ever possible, within a limited set of alternative sites. In this situation the execution of DM tasks typically involves the decision of how much data is to be moved and where. Also, summaries or other forms of aggregate information can be moved to allow more efficient transfers.

In other cases, data are produced locally but due to their huge volume cannot be stored in a single site and are therefore moved immediately after production to other storage locations, typically distributed on geographical scale. Examples are Earth Observing Systems

(EOS) [73], i.e. satellites sending their observational data to different earth stations, high energy physics experiments, that produce huge volumes of data for each event and send the data to remote laboratories for the analysis. In these cases, data can be replicated in more than one site and repositories can have a multi-tier hierarchical organization [22]. Problems of replica selection and caching management are typical in such scenarios.

The need for parallel and distributed architecture is not only driven by the data, but also by the high complexity of DM computations. Often the approach used by the DM analyst is exploratory, i.e. several strategies and parameter values are tested in order to obtain satisfactory results. Also, in many applications data are produced in streams that have to be processed on-line and in reasonable times with respect to the production rate of the data and of the specific application domain. Using high performance parallel and distributed architectures is therefore imperative.

4.1.2 DDM Systems

We give here some definitions of DDM systems, by analyzing three different approaches. They pose different problems and have different benefits. Existing DDM systems can in fact be classified in one of these approaches.

Data-driven. The simplest model for a DDM system only takes into account the distributed nature of data, but then relies on local and sequential DM technology. Since in this system the focus is solely posed in the location of data, we refer to this model as *data-driven*. In Figure 4.1 we sketch the data-driven architecture.

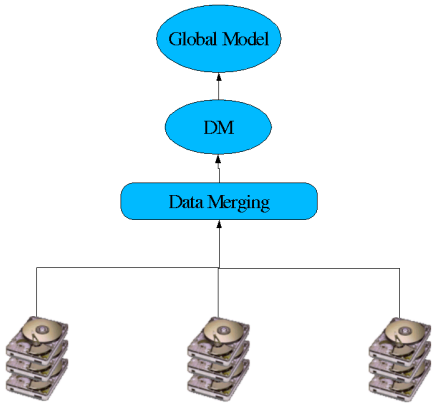


Figure 4.1: The data-driven approach for Distributed Data Mining systems.

In this model, data are located in different sites which do not need to have any computational capability. The only requirement is to be able to move the data to a central location in order to merge them and then apply sequential DM algorithms. The output of the DM analysis, i.e. the final knowledge models are then either delivered to the analyst' location or accessed locally where they have been computed.

The process of gathering data in general is not simply a merging step and depends on the original distribution. For example data can be partitioned horizontally – i.e. different records are placed in different sites – or vertically – i.e. different attributes of the same records are distributed across different sites. Also, the schema itself can be distributed, i.e. different tables can be placed at different sites. Therefore when gathering data it is necessary to adopt the proper merging strategy.

Although in this approach we can reuse much of the same DM technology available for the sequential case, a strong limitation is posed by the necessity of gathering all data in a single location. As already pointed out, this is in general unfeasible, for a variety of reasons.

Model-driven. A different approach is the one we call *model-driven*, depicted in Figure 4.2. Here, each portion of data is processed locally to its original location, in order to obtain partial results referred to as *local knowledge models*. Then the local models are gathered and combined together to obtain a global model.

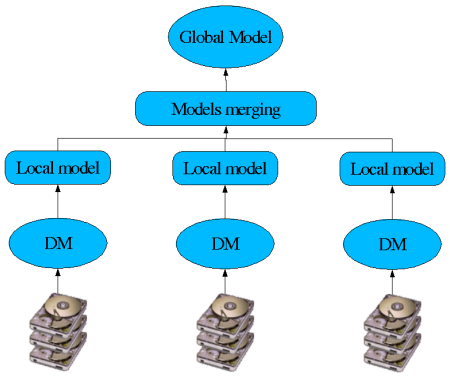


Figure 4.2: The model-driven model for Distributed Data Mining systems.

Also in this approach, for the local computations it is possible to reuse sequential DM algorithms, without any modification. The problem here is how to combine the partial results coming from the local models. Different techniques can be adopted, based on voting strategies or collective operations [50], for example. Multi-agent systems may apply meta-learning to combine partial results of distributed local classifiers [79]. The draw-back of the model-driven approach is than it is not always possible to obtain an exact final result, i.e. the global knowledge model obtained may be different from the one obtained by applying the data-driven approach (if possible) to the same data. Approximated results are not always a major concern, but it is important to be aware of that. Moreover, in this model hardware resource usage is not optimized. If the heavy computational part is always executed locally to data, when the same data is accessed concurrently, the benefits coming from the distributed environment might vanish due to the possible strong performance degradation.

The Papyrus system [36] is a DDM system that implements a cost-based heuristic to adopt either the data-driven or the model-driven approach.

Architecture-driven. In order to be able to control the performance of the DDM system, it is necessary to introduce a further layer between data and computation. As show in Figure 4.3, before starting the distributed computation, we consider the possibility of moving data to different sites with respect to where they are originally located, if this turns out to be profitable in terms of performances. Moreover, we introduce a communication layer among the local DM computations, so that the global knowledge model is built during the local computation. This allows for arbitrary precision to be achieved, at the price of a higher communication overhead. Since in this approach for DDM the focus is on optimized resource usage, we refer to this approach as the *architecture-driven*.

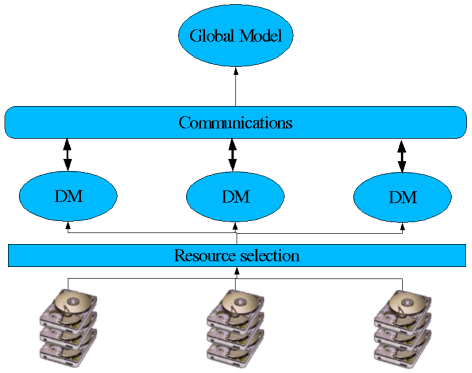


Figure 4.3: The architecture-driven approach for Distributed Data Mining systems.

The higher flexibility of this model and the potentially higher performance that it is possible to achieve, are payed in terms of the higher management effort that it is necessary to put in place. A suitable scheduling policy must be devised for the resource selection layer. Moreover, DM sequential algorithms are not reusable directly and must be modified or redesigned in order to take advantage of the communication channel among the different DM computations.

A notable example of such system is the Optimal Data and Model Partitions (OPT-DMP) framework [34] aimed at balancing performance and accuracy in DDM applications.

4.1.3 State of the art

Sequential DM, although always under development, is currently being applied in many application domains and it can be considered an *industry-ready* technology, ready and mature enough for delivery at industrial standards. The same does not hold for DDM, where research still plays a crucial role in finding solid results. Despite this stage of relative *infancy*, many important results have already been achieved. We review here the most important of them.

Parallel and Distributed DM Algorithms. The basic components of a DDM system are DM algorithms, able to mine valuable knowledge in a distributed environment. Many sequential clustering algorithms have been parallelized. In [12] [92] parallel implementation of the popular k-means algorithm [7] can be found. Talia *et al.* parallelized the unsupervised classification algorithm Autoclass [78].

Distributed Classifiers have been studied [79] able to operate either on homogeneous or heterogeneous data. They typically produce local models that are refined after exchanging summaries of data with the other classifiers.

A different approach is adopted in the Collective Data Mining framework [71], where instead of combining partial models coming from local analysis, globally significant pieces of information are extracted from local sites.

For example in Collective Principal Component Analysis (CPCA) [49], principal components are first found on local portion of the data. Then a sample of the data, projected on the local eigenvectors is exchanged among the distributed processes, PCA is performed in the reassembled data and the dominant eigenvectors found are transformed back to the original space.

In Distributed Association Rule Mining algorithms [102] we find the following major approaches.

Count distribution according to which the transaction database is statically partitioned among the processing nodes, while the candidate set C_k is replicated. At each iteration every node counts the occurrences of candidate itemsets within the local database partition. At the end of the counting phase, the replicated counters are aggregated, and every node builds the same set of frequent itemsets F_k. On the basis of the global knowledge of F_k, candidate set C_{k+1} for the next iteration is then built. Inter-Node communication is minimized at the price of carrying out redundant computations in parallel.

Data distribution attempts to utilize the aggregate main memory of the whole parallel system. Not only the transaction database, but also the candidate set C_k are partitioned in order to permit both kinds of partitions to fit into the main memory of each node. Processing nodes are arranged in a logical ring topology to exchange database partitions, since every node has to count the occurrences of its own candidate itemsets within the transactions of the whole database. Once all database partitions have been processed by each node, every node identifies the locally frequent itemsets and broadcasts them to all the other nodes in order to allow them to build the same set C_{k+1}. This approach clearly maximizes the use of node aggregate memory, but requires a very high communication bandwidth to transfer the whole dataset through the ring at each iteration.

Candidate distribution exploits problem domain knowledge in order to partition both the database and the candidate set in a way that allows each processor to proceed independently. The rationale of the approach is to identify, as execution progresses,

disjoint partitions of candidates supported by (possibly overlapping) subsets of different transactions. Candidates are subdivided on the basis of their prefixes. This is possible because candidates, frequent itemsets, and transactions, are stored in lexicographical order. Depending from the resulting candidate partitioning schema, the approach may suffer from poor load balancing. The parallelization technique is however very interesting. Once the partitioning schema for both C_k and F_k is decided, the approach does not involve further communications/synchronizations among the nodes.

The results of the experiments described in [5] demonstrate that algorithms based on Count Distribution exhibits optimal scale-up and excellent speedup, thus outperforming the other strategies. Data Distribution resulted the worst approach, while the algorithm based on Candidate Distribution obtained good performances but paid a high overhead caused by the need of redistributing the dataset.

We propose in Section 4.2 an hybrid strategy adopted in the parallelization of DCI where count-distribution is adopted for the first counting-based steps, and candidate-distribution for the intersection-based phase of the algorithm.

Distributed DM Systems. Many architectural issues are involved in the definition of full DDM systems.

Efficient communications are surely one of the main concerns. New protocols are being studied and proposed, like the Simple Available Bandwidth Utilization Library (SABUL) [39], which merges features of UPD and TCP to produce a light-weight protocol with flow control, rate control and reliable transmission mechanism, designed for data intensive applications over wide area high speed networks.

Other approaches try to optimize existing mechanisms for wide area data intensive applications. In [19], [96] optimizations for the GridFTP [8] file transfer utility for Grid systems [28], are studied.

One important issue in distributed systems is that of an efficient management of the resources available, namely a scheduler components that have to determine the *best* hardware/software resources to execute the DDM. The final target of such scheduler can be to reduce the volume of the communications altogether [36], or to minimize the overall completion time [59].

Another problem that it is worth mentioning is related to maintenance of the software components. In wide scale distributed environments the software components used to implement the DDM system might be several and undergo diverse evolution processes. The actual control of such components might not always be within the DDM system. Rather third-parties can let the DDM system use their components, but remain the only responsible for updating or changing them when needed. Efforts are being carried on toward the definition of standards for the interaction among such components, or for the standardization of the format used to describe the knowledge produced [38]. The *Predictive Model Markup Language* (PMML) [38], which provides the XML specification for several

kinds of data mining sources, models, and tools, also granting the interoperability among different PMML compliant tools, is the most interesting proposal to this regard.

There are also examples of complete DDM systems. JAM [79] and BODHI [51] are java based multi-agent systems. According to the classification proposed in Section 4.1.2 they follow the model-driven approach. In JAM each agent is placed in a different site, whereas in BODHI agents can move across different sites, together with their status and learned knowledge.

The previously mentioned Papyrus [36], is a systems that evaluates whether to move data, models or results. It is flexible in that is implements multiple strategies, but still lacks to take into account resource usage. The OPTDMP system [34], an architecture-driven system, addresses this problem. It adopts a linear model for the cost of a DDM computation. Its main innovative feature is to consider also the accuracy of the knowledge mined. It lacks, on the other hand, a satisfactory cost model for DM computations.

A complete architecture for large scale distributed datamining is called the Knowledge Grid (K-Grid), proposed in [93] by Talia *et al.*. Built on top of standard Grid middleware, like the Globus Toolkit [29], it defines services and components, specific to the DM and KDD domain. We will describe such framework in more detail in Section 4.3, where we study the scheduling of DM tasks onto the K-Grid.

4.1.4 Research Directions

As already stated, DDM technology needs to be further studied and enhanced in order to reach an industry standards. Such research efforts should be targeted at many levels.

There is the need for new distributed algorithms. Starting from the parallelization of existing sequential ones, it is necessary to develop new strategies for DM computations. One important feature that distributed DM algorithms should have is the ability of balancing accuracy and performance. In the distributed environment it might be better to find less accurate results that can be delivered faster to the final user, rather than determine exact results obtained in too long times.

Always more applications deal with streaming data that often require real-time executions. There is therefore the need for on-line algorithms, able to cope with the ever changing flow of input information.

An important issue that is becoming more studied in recent years, is that of the so called Privacy Preserving Data Mining [56] [6]. In distributed environments it might not be rare to be able to access to some data onto which privacy constraints hold. Distributed algorithm should be able to handle privacy issues.

Using different software components in distributed environment calls for modern technologies to face problems related to components maintenance and managements, from interface definition to model abstraction. An emerging technology to this regard is constituted by web services. A deep study of the possibilities offered by this technology to DDM could lead to interesting results.

Our efforts in the field of DDM are targeted at the definition of efficient parallel algorithms for association mining and at resource management issues in distributed environ-

ments. In the following of this Chapter, we describe ParDCI (Section 4.2) a parallel version of the already discussed DCI algorithm or Frequent Set Counting. In Section 4.3 we study the problems related to the scheduling of DM tasks on large scale distributed platforms. We propose an original cost model for DM tasks and a scheduling policy that properly takes into account data transfers and computations.

4.2 Parallel DCI

In ParDCI, we adopted different parallelization strategies for the two phases of DCI, i.e. the counting-based and the intersection-based ones. These strategies are slightly differentiated with respect to the two levels of parallelism exploited: *intra-node* level within each SMP to exploit shared-memory cooperation, and *inter-node* level among distinct SMPs, where message-passing cooperation is used. ParDCI uses *Count Distribution* during the counting-based phase, and a *Candidate Distribution* during the intersection-based phase [5, 40, 102]. The first technique requires dataset partitioning, replication of candidates and associated counters. The final values of the counters are derived by all-reducing the local counters. The latter technique is instead used during the intersection-based phase. It requires an intelligent partitioning of C_k based on the prefixes of itemsets, but a partial/complete replication of the dataset.

In the following we describe the different parallelization techniques exploited for the counting- and intersection-based phases of ParDCI, the parallel version of DCI. Since our target architecture is a cluster of SMP nodes, in both phases we distinguish between *intra-node* and *inter-node* levels of parallelism. At the inter-node level we used the message–passing paradigm through the MPI communication library, while at the intra-node level we exploited multi-threading through the *Posix Thread* library. A *Count Distribution* approach is adopted to parallelize the counting-based phase, while the intersection-based phase exploits a *Candidate Distribution* approach [5].

4.2.1 The counting-based phase

At the inter-node level, the dataset is statically split in a number of partitions equal to the number of SMP nodes available. The size of partitions depend on the relative powers of nodes. At each iteration k, each node independently generates an identical copy of C_k. Then each node p reads blocks of transactions from its own dataset partition $D_{p,k}$, performs subset counting, and writes pruned transactions to $D_{p,k+1}$. At the end of the iteration, an all-reduce operation is performed to update the counters associated to all candidates of C_k, and all the nodes produce an identical copy of F_k.

At the intra-node level, each node runs a pool of threads. They have the task of checking in parallel candidate itemsets against chunks of transactions read from $D_{p,k}$. The task of subdividing the local dataset into disjoint chunks is assigned to a particular thread, the *Master Thread*. It loops reading blocks of transactions and forwarding them to the *Worker Threads* executing the counting task. To overlap computation with I/O, minimize

synchronization, and avoid unnecessary data copying overheads, we used an optimized producer/consumer schema for the cooperation among the Master and Worker threads. Through two shared queues, the Master and Worker threads cooperate by exchanging pointers to empty and full buffers storing blocks of transactions to be processed.

When all transactions in $D_{p,k}$ have been processed by node p, the corresponding Master thread performs a local reduction operation over the thread-private counters (reduction at the intra-node level), before performing via MPI the global counter reduction operation with all the other Master threads running on the other nodes (reduction at the inter-node level). Finally, to complete the iteration of the algorithm, each Master thread generates and writes F_k to the local disk.

4.2.2 The intersection-based phase

During the intersection-based phase, a Candidate Distribution approach is adopted at both the inter- and intra-node levels. This parallelization schema makes the parallel nodes completely independent: inter-node communications are no longer needed for all the following iterations of ParDCI.

Let us first consider the inter-node level, and suppose that the intersection-based phase is started at iteration $\overline{k}+1$. Therefore, at iteration $\overline{k}$ the nodes build on-the-fly the bit-vectors representing their own in-core portions of the vertical dataset. Before starting the intersection-based phase, each node broadcasts its own partial vertical dataset to all the other nodes in order to obtain a complete replication of the whole vertical dataset on each node.

Dataset partitions are typically quite large, and depending on the capabilities of the interconnection network, their broadcast may involve a significant communication overhead. Moreover, common distributions of MPI such as MPICH and LAM/MPI, rely on unoptimized implementations of collective operations.

In order to minimize the performance penalty due to the inefficiency of current MPI implementations, we designed a more efficient broadcasting algorithm. According to our algorithm:

1. before broadcasting its own partial vertical dataset, each node compresses it by means of the *Zlib* data-compression library. The fastest *Zlib* compression level is used, resulting in a negligible compression time, and in a size of the compressed dataset partition that is in the average about one third of the uncompressed one;

2. to minimize network congestion, the *all-to-all* broadcast is then performed in $(P-1)$ communication steps, with P number of processing nodes involved. At each step the nodes are split into $P/2$ disjoint sets, so that all possible node pairs are considered. During each communication step, the nodes of each pair exchange their own compressed partitions in parallel, decompress the partition and store it in the buffer allocated for holding the global vertical dataset.

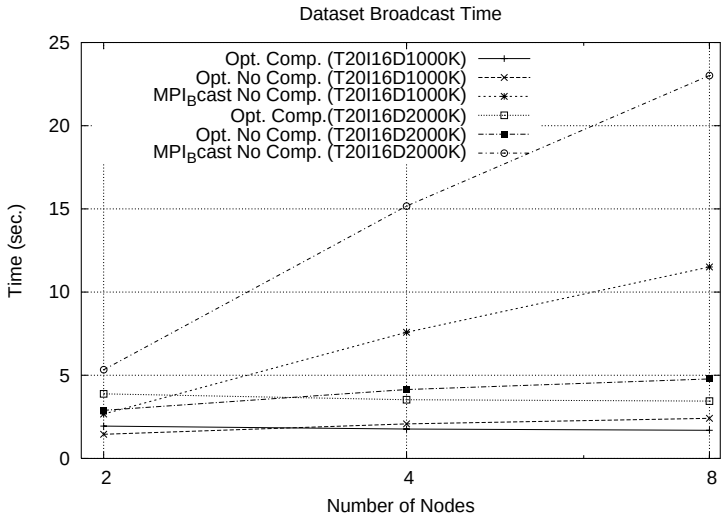


Figure 4.4: Elapsed times required to broadcast the vertical dataset with our optimized broadcast algorithm (**Opt.**) exploiting or not compression, and the native **MPI_Bcast** routine.

Figure 4.4 compares the elapsed times of our optimized broadcast algorithm (**Opt.**), exploiting or not compression (**Comp.** or **No Comp.**), with those obtained using the MPI native broadcast routine **MPI_Bcast**. The measures refer to the execution of ParDCI on a cluster of linux workstations with a switched FastEthernet network. We used two synthetic datasets and ran the tests on 2, 4, and 8 nodes. The size of the vertical dataset broadcast is about 28MB for dataset T20I16D1000K, and 57 MB for dataset T20I16D2000K. When the compression is exploited, the sizes of the same vertical datasets decrease to about 10MB and 20MB, respectively. In this case, the broadcast times reported include also the time required to compress and decompress data. As it can be seen, our algorithm scales well and results in broadcast times remarkably lower than those obtained by using the native MPI broadcast routine. Moreover, compression results to be advantageous when more than two processing nodes are used.

Once the global vertical dataset is built on each node, the frequent set $F_{\overline{k}}$ computed in the last counting-based iteration is partitioned, and a disjoint partition $F_{p,\overline{k}}$ of $F_{\overline{k}}$ is assigned to each node p, where $\bigcup_p F_{p,\overline{k}} = F_{\overline{k}}$.

Such partitioning entails a Candidate Distribution parallelization schema, according to which each node p will be able to generate a unique F_k^p $(k > \overline{k})$ independently of all the other nodes, where $F_k^p \cap F_k^{p'} = \emptyset$ if $p \neq p'$, and $\bigcup_p F_k^p = F_k$. Candidate Distribution parallelization strategies can severely suffer load imbalance. We thus paid particular attention to design the strategy adopted to partition $F_{\overline{k}}$.

Our partitioning strategy works as follows. First, $F_{\overline{k}}$ is split into l *sections* on the basis of the prefixes of the lexicographically ordered frequent itemsets included. All the frequent $\overline{k}$-itemsets that share the same $\overline{k}-1$ prefix are assigned to the same section. Since candidate $(\overline{k}+1)$-itemsets are generated as the union of two frequent $\overline{k}$-itemsets sharing

the first $\overline{k}-1$ items, partitioning according to the $\overline{k}-1$ prefixes assures that all candidates can be generated starting from one (and only one) of the l disjoint sections of $F_{\overline{k}}$. Clearly, the same holds for all $k > \overline{k}$ as well.

The $F_{p,\overline{k}}$ partitions are then created by assigning the l sections to the processing nodes with a simple greedy strategy that considers a weight associated to each section. The weight is an estimate of the computational cost associated with the section. Such cost is dominated by the cost of the intersections, which is on turn related to the number of candidates that will be generated from the itemsets in the section.

ParDCI uses two different methods to estimate the computational cost of a section. When the number of different prefixes is large with respect to the number P of processing nodes used, the weight associated to each section is simply the number of itemsets belonging to the section itself. In fact, when $l \gg P$ holds, we can expect to obtain a good load balancing also adopting such a trivial method. Otherwise, when the number of sections is not very large (as often happens when dense datasets are mined), it is necessary to assign a weight proportional to the expected computational cost for the section.

The basic idea is that the candidates obtained from a section are related to the *order* of the dataset partition constituted by the tidlists of the distinct items present in the section. The more ordered the partition is, the more candidates candidates will be found.

Following an analysis similar to the one illustrated in Section 2.5, we assigned to each section a weight proportional to the quantity H_1, measured for the corresponding section.

Once completed the partitioning of $F_{\overline{k}}$, the nodes independently generate the associated candidates and determine their supports by intersecting the corresponding bit-vectors. Nodes continue to work according to the schema above also for the following iterations, without any communication exchange.

At the intra-node level, a similar Candidate Distribution approach is employed, but at a finer granularity by using dynamic scheduling to ensure load balancing among the threads. In particular, at each iteration k the Master thread of a node p initially splits the local partition of $F_{p,k-1}$ into x disjoint partitions S_i, $i = 1, \ldots, x$, where $x \gg t$, and t is the number of active threads. The boundaries of these x partitions are then inserted in a shared queue. Once the shared queue is initialized, also the Master thread becomes a Worker. Hereinafter, each Worker thread loops and self-schedules its work by performing the following steps:

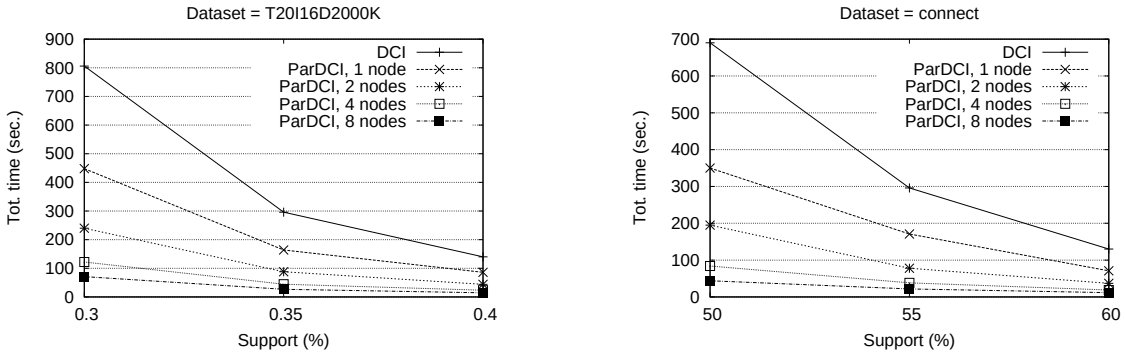
1. access in mutual exclusion the queue and extract information to get S_i, i.e. a partition of the local $F_{p,k-1}$. If the queue is empty, write $F_{p,k}$ to disk and start a new iteration.

2. generate a new candidate k-itemset c from S_i. If it is not possible to generate further candidates, go to step 1.

3. compute on-the-fly the support of c by intersecting the vectors associated to the k items of c. In order to reuse effectively previous work, each thread exploits a private cache for storing the partial results of intersections (see Section 2.4.4). If c turns out to be frequent, put c into $F_{p,k}$. Go to step 2.

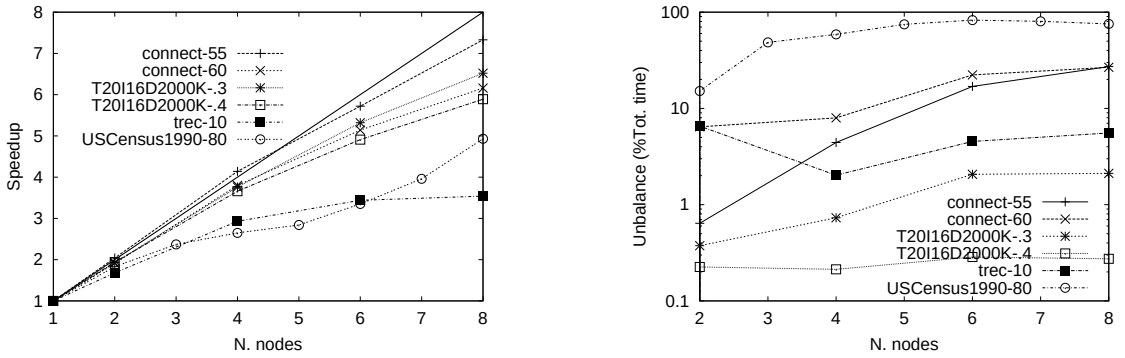
4.2.3 Performance evaluation of ParDCI

We evaluated ParDCI on both dense and sparse datasets. Figure 4.5 plots the total execution time required by ParDCI to mine datasets `T20I16D2000K` and `connect-4` as a function of the support threshold s. The curves refer to the execution of ParDCI on 1, 2, 4, and 8 processing nodes. The pure multi-threaded version of ParDCI is used for the tests with a single SMP node, while in the tests with 2, 4, and 8 nodes ParDCI also exploits inter-node parallelism.



(a) (b)

Figure 4.5: Datasets `T20I16D2000K` (a) and `connect-4` (b): ParDCI execution times on 1, 2, 4, and 8 processing nodes, varying the minimum support threshold.



(a) (b)

Figure 4.6: Speedups (a) and relative load imbalance (b) achieved by running ParDCI on various datasets.

Figure 4.6 plots the speedups obtained on various datasets as a function of the number of SMP nodes used. As it can be seen, the speedups achieved are in most cases good, with

efficiencies higher than 0.75. Only for the USBCensus1990 dataset the speedup resulted remarkably lower than in the other cases.

The reason of this lower scalability is evidenced by Figure 4.6.(b) that plots the differences in percentage over the total execution time between the first and last node to end execution. The poor load balancing achieved on the `USCensus1990` dataset is due the Candidate Distribution phase of the algorithm: in fact, more than the 50% of the total candidates mined originates from only 3 sections of $F_{\overline{k}}$. Given such an unusual candidate distribution, balancing well the load becomes impossible. On the other hand, we could obtain a finer candidate partitioning by forcing DCI to start its intersection-based phase later. Even if this trick allows DCI to obtain a good load balance, the total execution time become remarkably worse, due to the cost of performing an additional expensive count-based iteration. In all the other cases, we observed a limited imbalance, thus demonstrating that the strategies adopted for partitioning dataset and candidates on our homogeneous cluster of SMPs are effective.

4.3 Scheduling DM tasks on distributed platforms

The distributed nature of data already discussed in this Chapter and the need for high performance, make large scale distributed environments, like the Grid [28], a suitable environment for DDM [89]. Grids may in fact provide coordinated resource sharing, collaborative processing, and high performance computing platforms. Since DDM applications are typically *data intensive*, one of the main requirements of such a DDM Grid environment is the efficient management of storage and communication resources.

In the rest of this Chapter we are going to introduce the problem of scheduling DM tasks on large scale distributed platforms provided with special purpose facilities for DDM, namely the Knowledge Grid (K-Grid) [93]. One of the main problems of grid schedulers for DM, is related to cost modeling for DM tasks. We propose an architecture for a K-Grid scheduler that adopts a sampling based original technique for performance prediction.

4.3.1 Data and Knowledge Grid

A significant contribution in supporting data intensive applications is currently pursued within the Data Grid effort [22], where a data management architecture based on storage systems and metadata management services is provided. The data considered here are produced by several scientific laboratories geographically distributed among several institutions and countries. Data Grid services are built on top of Globus [29], a middleware for Grid platforms, and simplify the task of managing computations that access distributed and large data sources.

The Data Grid framework share most of its requirements with the realization of a Grid-based DDM system, where data involved may originate from a larger variety of sources. Even if the Data Grid project is not explicitly concerned with data mining issues, its basic services could be exploited and extended to implement higher level grid services dealing

with the process of discovering knowledge from larger and distributed data repositories. Motivated by these considerations, in [93] a specialized grid infrastructure named *Knowledge Grid* (K-Grid) has been proposed (Figure 4.7). This architecture was designed to be compatible with lower-level grid mechanisms and also with the Data Grid ones. The authors subdivide the K-Grid architecture into two layers: the core K-grid and the high level K-grid services. The former layer refers to services directly implemented on the top of generic grid services, the latter refers to services used to describe, develop and execute parallel and distributed knowledge discovery (PDKD) computations on the K-Grid. Moreover, the layer offers services to store and analyze the discovered knowledge.

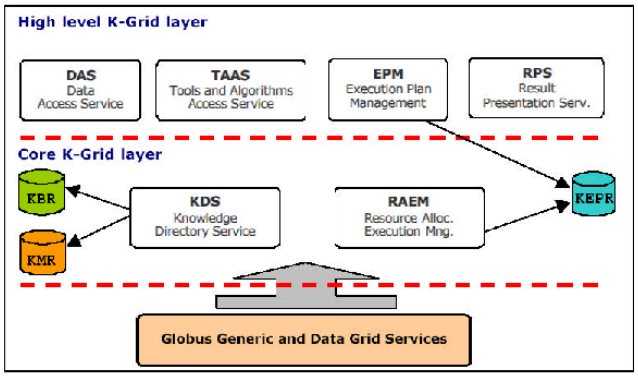


Figure 4.7: General scheme of the Knowledge Grid software architecture. This picture is taken from [93].

We concentrate our attention on the K-Grid core services, i.e. the Knowledge Directory Service (KDS) and the Resource Allocation and Execution Management (RAEM) services. The KDS extends the basic Globus Metacomputer Directory Service (MDS) [27], and is responsible for maintaining a description of all the data and tools used in the K-Grid. The metadata managed by the KDS are represented through XML documents stored in the Knowledge Metadata Repository (KMR). Metadata regard the following kind of objects: data sources characteristics, data management tools, data mining tools, mined data, and data visualization tools.

Metadata representation for output mined data models may also adopt the (PMML) [38] standard, already mentioned in Section 4.1.3.

The RAEM service provides a specialized *broker* of Grid resources for DDM computations: given a user request for performing a DM analysis, the broker takes allocation and scheduling decisions, and builds the *execution plan*, establishing the sequence of actions that have to be performed in order to prepare execution (e.g., resource allocation, data and code deployment), actually execute the task, and return the results to the user. The execution plan has to satisfy given requirements (such as performance, response time, and mining algorithm) and constraints (such as data locations, available computing power, storage size, memory, network bandwidth and latency). Once the execution plan is built, it is passed to the Grid Resource Management service for execution. Clearly, many differ-

ent execution plans can be devised, and the RAEM service has to choose the one which maximizes or minimizes some metrics of interest (e.g. throughput, average service time).

4.3.2 The Knowledge Grid Scheduler

K-Grid services can be used to construct complex Problem Solving Environments, which exploit DM kernels as basic software components that can be applied one after the other, in a modular way. For example we can perform an initial clustering on a given dataset in order to extract groups of *homogeneous* records, and then look for association rules within each cluster. An example of such system is the VEGA visual tool [20] for composing and executing DM applications on the K-Grid.

A general DM task on the K-Grid can therefore be described as a Directed Acyclic Graph (DAG) whose nodes are the DM algorithms being applied, and the links represent data dependencies among the components. K-Grid users interact with the K-Grid by composing and submitting DAGs, i.e. the application of a set of DM kernels on a set of datasets.

In this scenario, one important service of the K-Grid is the one that is in charge of mapping task requests onto physical resources. The user will in fact have a transparent view of the system and possibly little or no knowledge of of the physical resources where the computations will be executed, neither he or she knows where the data actually reside. The only thing the user must be concerned with, is the semantic of the application, i.e. what kind of analysis he or she wants to perform and on which data.

4.3.3 Requirements

Many efforts have already been devoted to the problem of scheduling distributed jobs in general [88], [90] and for Grid platform in particular, like Nimrod-g [1], Condor [24] and AppLeS [15]. Recently a general architecture for grid schedulers has been outlined in [85] by J. Schopf. She describes three main phases of the activity of a grid scheduler. The first phase is devoted to resource discovery. During this phase a set of candidate machines where the application can be executed, is built. This set is obtained by filtering the machines where the user has enough privileges to access and at the same time satisfy some minimum requirements, expressed by the user. In the second phase one specific resource is selected among the ones determined in the previous phase. This choice is performed based in information about system status - e.g. machine loads, network traffic - and again possible user requirements in term of execution deadline or limited budget. Finally, the third phase is devoted to actual job execution, from reservation to completion, though submission and monitoring.

The second phase described above is the most challenging, since it is strictly application dependent. Many of the schedulers mentioned above propose their own solution to the problem. Nevertheless, there are some characteristics of scheduling DM tasks, that make inadequate the previous approaches.

First of all we lack an accurate analytical cost model for DM tasks. In the case of the Nimrod-g system, the parametric, exactly known cost of each job allows the system to foresee with a high degree of accuracy which is going to be the execution time of each job. This does not hold for DM, where the execution time of an algorithm in general depend on the input parameters in a non linear way, and also on the dataset internal correlations, so that, given the same algorithm, the same set of parameters and two dataset of identical dimensions, the execution time can vary of orders of magnitude. The same can be said for other performance metrics, as memory requirement and I/O activity.

The other characteristic is that scheduling a DM task in general implies scheduling computation and data transfer [74]. Traditional schedulers typically only address the first problem, that of scheduling computations. In the case of DM, since the dataset are typically *big*, it is also necessary to properly take into account the time needed to transfer data and to consider when and if it is worth to move data to a different location in order to optimize resource usage or overall completion time.

We resume the characteristics of the scheduler we want to design, for mapping DM tasks on the K-Grid. We call such scheduler the Knowledge Grid Scheduler, or KGS.

On-line. KGS must schedule the components of DM DAGs as soon as they arrive in the system.

Dynamic. It must apply cost models to predict future resource status and pro-actively assign jobs to resources.

Adaptive. It must continuously interact with the K-Grid Information Service, in order to have an updated view of the system status in terms of machine and network loads.

4.3.4 Design of KGS

We describe here the design of KGS. A model for the resources of the K-Grid, depicted in Figure 4.8 is composed by a set of hosts, onto which the DM tasks are executed, a network connecting the hosts and a centralized scheduler, KGS, where all requests arrive.

The first step is that of task composition. We do not actually deal with this phase and we only mention it here for completeness. As explained earlier, we consider that the basic *building blocks* of a DM task are algorithms and datasets. As shown in Figure 4.9 they can be combined in a structured way, thus forming a DAG.

DM components correspond to a particular algorithm to be executed on a given dataset, provided a certain set of input parameters for the algorithm. We can therefore describe each DM components Λ with the triple:

$$\Lambda = (A, D, \{P\}) \tag{4.1}$$

where A is the data mining algorithm, D is the input dataset, and $\{P\}$ is the set of algorithm parameters. For example if A corresponds to "Association Mining", then $\{P\}$ could be the minimum confidence for a discovered rule to be meaningful. It is important

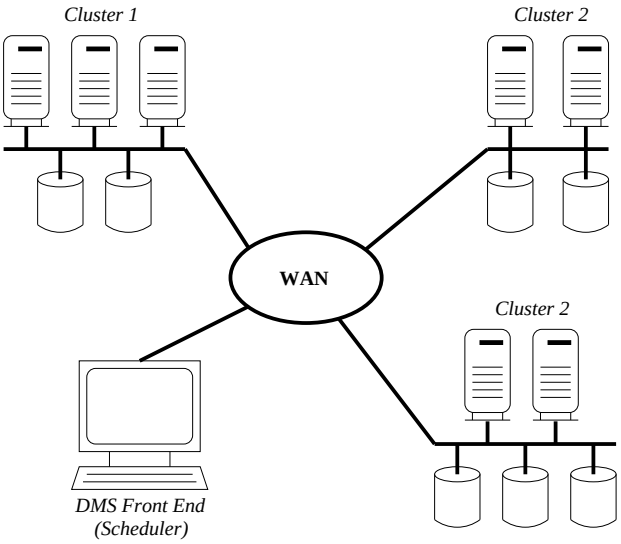


Figure 4.8: Physical resources in the K-Grid.

to notice that A does not refer to a specific implementation. We could therefore have more different implementations for the same algorithm, so that the scheduler should take into account a multiplicity of choices among different algorithms and different implementations. The best choice could be chosen considering the current system status, the programs availability and implementation compatibility with different architectures.

Choosing the set of resources onto which the DAG can be mapped, corresponds to a modification of the original DAG according to the features of the resources chosen. In Figure 4.10 we schematically represent how such modification takes place.

The original DM task on the left hand side, is composed by the application of a first clustering algorithm on a certain dataset, and then by the application of an algorithm for association mining on each cluster found. Finally all the results are gathered for visualization. In the mapping reported on the right hand side of Figure 4.10, we add a node to the top of the graph, which corresponds to the initial determination of the input dataset. Moreover, we detail the structure of the actual computation performed when we chose a specific implementation for each software component. In Figure 4.10 we reported the case where one of the association rules mining is performed on a parallel architecture, and its structure is therefore shown.

In this way, starting from a *semantic* DAG, we define a *physical* DAG, derived from the first one, with all the components mapped onto actual physical resources.

This process is repeated for all the DAGs that arrive at the scheduler.

The global vision of the system is summarized in Figure 4.11. The first step is the creation of the semantic DAGs from the basic components. This step is in general performed by several users at the same time. Therefore we have a burst of DAGs that must be mapped on the system. Semantic DAGs queue in scheduler and wait their turn. When a DAG is processed, the scheduler build the physical and determine the best set of resources

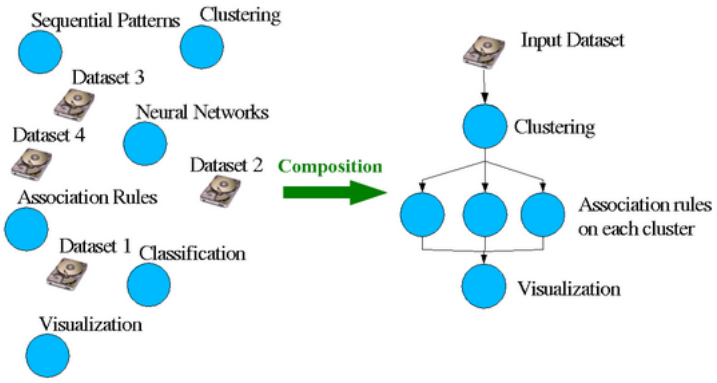


Figure 4.9: Composition of a DM DAG in terms of basic building blocks: datasets and algorithms.

where the DAG can be mapped. This is done by taking into account current system status, i.e. network and machines load as induced by previous mappings, and also by verifying that all data dependencies are satisfied. Referring to the example above, the scheduler must first schedule the clustering algorithm and then the association mining.

Scheduling DAGs on a distributed platform is a non-trivial problem which has been faced by a number of algorithms in the past. See [55] for a review of them. Although it is crucial to take into account data dependencies among the different components of the DAGs present in the system, we first want to concentrate ourselves on the cost model for DM tasks and on the problem of bringing communication costs into the scheduling policy. For this reason, we introduce in the system an additional component that we call *serializer* (Figure 4.12), whose purpose is to decompose the tasks in the DAG into a series of independent tasks, and send them to the scheduler queue as soon as they become executable w.r.t. the DAG dependencies.

Such serialization process is not trivial at all and leave many important problems opened, such as determine the best ordering among tasks in a DAG that preserve data dependencies and minimizes execution time.

Nevertheless, at this stage of the analysis, as already stated, we are mainly concerned with other aspects in the system, namely the definition of an accurate cost model for single DM tasks and the inclusion of communications into the scheduling policy.

In the following we give a more accurate description of the mapping process, starting from the definition of models for the system architecture, the cost of DM tasks, and their the execution.

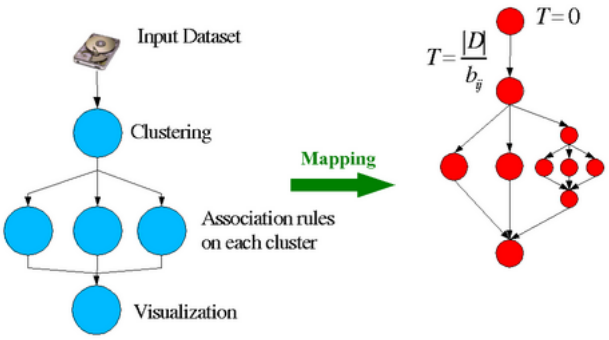


Figure 4.10: Mapping of a DAG onto physical resources permits to obtain cost measures for the DM task associated to the DAG.

4.3.5 Architecture model

The K-Grid architecture is composed by a set of clusters. Every host has a storage unit and contains a subset of the datasets available in the system. Dataset in general may be replicated at different sites. We assume that every host has the complete set of all available algorithms in the system, so that it is only necessary to move data and not algorithms.

Hosts are connected by fast link in Local Area Networks (LANs), thereby forming *clusters*. Slower links of a geographical Wide Area Network (WAN), connect the clusters.

We assume the scheduler is able to have a complete image of the system at any time, in particular it is aware of the load on each host and of the network status. The scheduler is, logically, the system front-end to the users, and provides them a coherent view of the system. From the front-end, users can choose a dataset and a DM task to be performed on the dataset, among a list of possibilities. A user request generates a job submission to the scheduler, which in turn will locate the data, chose the best algorithm implementation and will assign the task to a specific host or set of hosts. At this level we assume that hosts are all locally controlled by batch queues that allow only one job to be executed at the time.

Modeling networks is in general a nontrivial issue, due to the high complexity of the system to be described and to the many factors that cooperate in determining a single event: bit packets transfer from one location to another one in the network. When considering geographical networks, the complexity is so high to vanish most of the modeling efforts [75]. In our system, network links are the channel for dataset transfer, where the dataset range typically from hundreds of MB to tens of GB. Therefore we are not interested in the single IP packet, but rather in the overall link capacity when transmitting big datasets. For this reason we assume that the Wide Area Network (WAN) connecting the clusters can be described by a fixed bandwidth pipe. The bandwidth we refer to, is an experimental measure of the effective link capacity when transferring *big* files, without

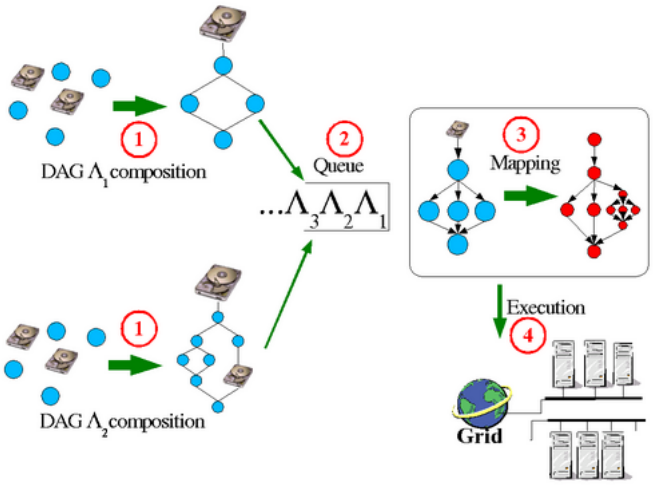


Figure 4.11: The whole system

taking into account all the physical details.

Similar considerations also motivated the architecture proposed in [96] to obtain an experimental measure of the bandwidth experimented in wide area data transfers, based on past transfers log information and current network status. The errors made by these predictions are still non-negligible, but in most cases it is possible to have a reasonable estimate on the transfer time.

LAN and WAN links are therefore described by a function of the number of the communications taking place.

For WAN links transmitting *big* datasets, we can assume a constant bandwidth up to a given number of connections, after that, the effective measured bandwidth starts decreasing. If f_{ij} is the function describing the effective measured bandwidth of the link between nodes i and j, and n is the number of files being transferred between i and j, we choose f as

$$f_{ij}(n) = \begin{cases} b & \text{if } n \leq \hat{n} \\ b\frac{\hat{n}}{n} & \text{if } n > \hat{n} \end{cases} \tag{4.2}$$

where b is the effective bandwidth and $\hat{n}$ is the maximum number of connections that preserve the service time.

In Figure 4.13 we report the bandwidth experimented when transferring a 16 MBytes file across three links connecting three sites: Pisa-Venice (PI-VE), Pisa-Cosenza (PI-CS) and Cosenza-Venice (VE-CS). We plot the bandwidth measured every hour, along a time interval of one week, for a single, double and triple stream transfer. The simulation started on a Tuesday morning at 9 am.

As it can be seen, there are essentially two different regimes: the *unloaded* one, characterized by a bandwidth of about 1Mbps with small fluctuations, which is essentially experimented at night ad during the weekend. The other regime is characterized by a lower

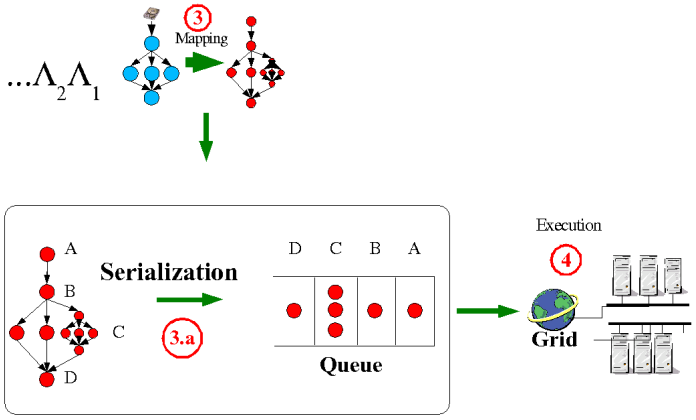


Figure 4.12: We simplify the DAG-based description of DM applications in order to be able to test the cost model for DM tasks and the scheduling policy.

average bandwidth, about 0.5Mbps, with bigger fluctuations. We can therefore adjust the model proposed by considering two different values for b_{ij} according to the particular time of the day at which the transfer is taking place.

We resume the global system architecture in Figure 4.14.

We plotted only the entities that are relevant for scheduling, in terms of queuing network. This description of the system will be useful when describing the scheduling policy. The entities represented in Figure 4.14 are the hosts, which are described by means of batch queues Q_i, one for each node, network links, described by shared service centers with no queue and a service time which is function of the source, the destination and of the number of connections already present with the same source and destination. Finally the scheduler itself is represented. Its internal structure is composed by three queues, whose behavior will be discussed in the Section 4.3.7.

4.3.6 Cost model

The critical part of our scheduler is the cost model adopted for the execution of DM tasks. In [54] a cost model is proposed for distributed DM algorithms. The authors propose a classification of distributed DM systems, which can be either *client-server* or *agent* based. They find cost models for both approaches and determine a heuristic to chose which one is the most convenient. They make the strong assumption that the actual cost of the DM task is known in advance. Another limit of their analysis is that data transfers are not taken into account properly.

In [46] [16] the memory access patterns of DM applications are studied in order to find optimization strategies and a characterization of such patterns. Among other considerations, the authors pointed out that most of the results do not depend on the number of records of the input dataset.

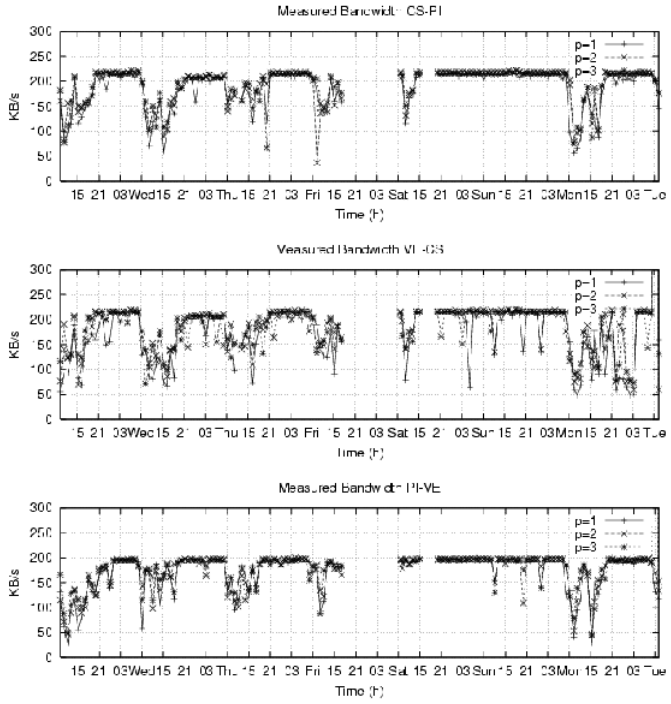


Figure 4.13: The measured bandwidth for the transfer of a 16MB dataset between Pisa and Venice (bottom), Pisa and Cosenza (top), Cosenza and Venice (middle), at different hour of the day, during a period of one week, starting from a Tuesday morning at 9 am.

The model we adopt in our system is the following. We suppose that when jobs arrive in the system we ignore their actual cost in terms of memory required and total execution time. For each task, we consider two possible alternatives: either to execute it sequentially, or to run it in parallel. In both cases we have to consider the cost of transferring the input dataset if the target host does not hold a copy of the data.

For both cases the general form for the total execution time of the data mining job, can be written as the sum of three terms:

$$T = T_{input} + T_{DM} + T_{output} \tag{4.3}$$

where T_{input} is the time needed for transferring the input dataset from the location where it is stored, to the machine where it will be executed, T_{DM} is the actual cost of the data mining task on that specific machine, and T_{output} is the time needed for transferring

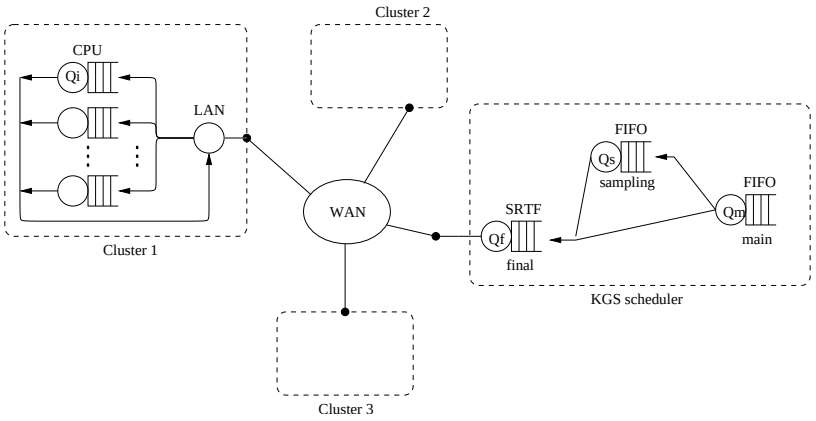


Figure 4.14: The model of the complete system, including computational resources (clusters), network links and the scheduler.

the output to the final location, which at this level we assume to be the front end location.

As explained before, we can work out T_{input} and T_{output} once we know the effective bandwidth among the nodes and the size of the dataset being transferred.

In the case of a sequential execution, the total execution time needed to complete job $\Lambda(A, D, \{P\})$ on host i given the input dataset D is stored on host h and the result has to be delivered at site k, is thus:

$$T_i^\Lambda = \frac{|D|}{b_{hi}} + \tau_i^\Lambda + \frac{|\alpha^\Lambda D|}{b_{ik}} \tag{4.4}$$

where τ_i^Λ is the cost of the data mining task and $|\alpha^\Lambda D|$ is the size of the output produced by the algorithm. Of course, the relative communication costs involved in dataset movements are zeroed if either $h = i$ or $h = k$.

We now consider the cost of the parallel execution. In this case the input dataset has to be partitioned among the nodes of the cluster c_j. The total execution time can be therefore written as:

$$T_j^\Lambda = \sum_{i \in c_j} \frac{|D|/|c_j|}{b_{hi}} + \tau_j^\Lambda + \sum_{i \in c_j} \frac{|\alpha^\Lambda(D)|/|c_j|}{b_{ik}} \tag{4.5}$$

Of course, the relative communication costs are zeroed if the dataset is already distributed, and is allocated on the nodes of c_j.

If we assume that all the parallel algorithms have an ideal behavior, and their speedup is optimal, we can say that the parallel execution time for the data mining task on cluster j, i.e. τ_j^Λ, is given by the execution time of the same task on a single node of the cluster, divided by the number of nodes in the cluster:

$$\tau_j^\Lambda = \frac{\tau_h^\Lambda}{|c_j|}. \tag{4.6}$$

We therefore have a cost model which is based on the knowledge of the sequential implementation of the algorithm. The problem is how to determine this cost. We propose to use sampling to obtain this information. Traditionally, sampling has been used as an approximation to the knowledge that can be extracted from a dataset by a DM algorithm [107] [80]. Here we are instead using sampling as a method for performance prediction.

The hypotheses behind this idea is that the relevant performance factors like total execution time or the amount of memory required, in most cases scale linearly with the input dataset size.

Based on this assumption, the cost of a data mining job Λ is given by the execution time of the same job on a sample of the input dataset. If $s \in [0, 1]$ is the sampling rate, we can write:

$$\tau_i^{\Lambda} = \frac{\tau_i^{\Lambda_s}}{s} \tag{4.7}$$

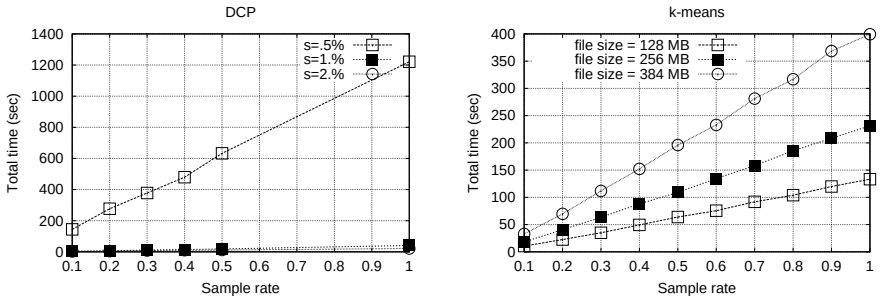
where $\Lambda_s = \Lambda(A, Ds, \{P\})$. The application of this idea is therefore to pre-execute the data mining task on a small sample of the original dataset, and to forecast the actual job duration based on this measurement.

The last factor we need to take into account is the different processor performances. We assume that such differences can be captured by a unique performance factor, so that the execution time of job Λ on a generic host h is given by:

$$\tau_h^{\Lambda} = \frac{\tau_i^{\Lambda_s}}{s} p_{ih} \tag{4.8}$$

where p_{ih} is the relative performance of nodes i and h.

The performance information coming from sampling can also be reused for more than one job request. Therefore, while sampling jobs are being executed, the scheduler can build a knowledge model to use for performance prediction.



(a) (b)

Figure 4.15: Execution time of the DCI FSC algorithm (a), and the k-means clustering one (b), as a function of the sample rate of the input dataset.

In order to validate the sampling-based method for performance prediction, we analyzed two DM algorithms: the counting based phase [67] of DCI (see Chapter 2) and *k-means* [12]. We ran DCI and *k-means* on synthetic datasets by varying the size of the sample considered. The results of the experiments are promising: both algorithms exhibit quasi linear scalability with respect to the size of the sample of a given dataset, when user parameters are fixed. Figure 4.15.(a) reports the DCI completion times on a dataset of medium size (about 40 MB) as a function of the size of the sample, for different user parameters (namely the minimum support $s\%$ of frequent itemsets). Similarly, in Figure 4.15.(b) the completion time of *k-means* is reported for different datasets, but for identical user parameters (i.e., the number k of clusters to look for). The results obtained for other datasets and other user parameters are similar, and are not reported here for sake of brevity. Note that the slopes of the various linear curves depend on both the specific user parameters and the features of the input dataset D. Therefore, given a dataset and the parameters for executing one of these DM algorithms, the slope of each curve can be captured by running the same algorithm on a smaller sampled dataset $\widehat{D}$.

For other algorithms, scalability curves may be more complex than a simple linear one. For example when the dataset size has a strong impact on the in-core or out-core behavior of an algorithm, or on the main memory occupation. So, in order to derive an accurate performance model for a given algorithm, it should be important to perform an off-line training of the model, for different dataset characteristics and different parameter sets.

Another problem that may occur in some DM algorithms, is the generation of false patterns for small sampling sizes. In fact, according to [107], we found that the performance estimation for very small sampling sizes may overestimate the actual execution times on the complete datasets. An open question is to understand the impact of this overestimation in our Grid scheduling environment.

4.3.7 Execution model and scheduling policy

We now describe how this cost model can be used by a scheduler that receives a list of jobs to be executed on the K-Grid, and has to decide for each of them which is the best resource to start the execution on.

Choosing the best resource implies the definition of a scheduling policy, targeted at the optimization of some metric. One frequent choice [88] is to minimize the completion time of each job. This is done by taking into account the actual ready time for the machine that will execute the job - remember that machines are locally controlled by batch queues - and the cost of execution on that machine, plus the communications needed. Therefore for each job, the scheduler will chose the machine that will finish the job earlier. For this reason in the following we refer to such policy as Minimum Completion Time (MCT).

Jobs $\Lambda(A, D, \{P\})$ arrive at the scheduler from an external source, with given timestamps. They queue in the scheduler and wait. We assume the jobs have no dependencies among one another and their inter-arrival time is given by an exponential distribution. When jobs arrive they have a sampling flag turned on. Such flag tells the scheduler that the job is currently of unknown duration. If the scheduler does not have any performance

measure for that job, given by previous execution of another similar job, then it generates a smaller sampling job.

The scheduler internal structure can be modeled as a network of three queues, plotted in Figure 4.14, with different policies.

Jobs arrive in the main queue Q_m, where sampling jobs are generated and appended to the sampling queue Q_s. Both queues are managed with a FIFO policy. From Q_s jobs are inserted into the system for execution. Once sampling is completed, the job in inserted in the final queue Q_f, where it is processed for the real execution. Since the scheduler knows the duration of jobs in Q_f, due to the prior sampling, Q_f is managed with a Shortest Remaining Time First (SRTF) policy in order to avoid light (*interactive*) jobs being blocked by heavier (*batch*) jobs.

The life-cycle of a job in the system is the following:

1. Jobs arrive in the main queue Q_m with a given timestamp. They are processed with FIFO policy. When a job is processed, the scheduler generates a sampling job and put this request in the sampling queue, with the same timestamp and a sampling rate equal to s_0, equal for all jobs.

2. If a job in Q_m has parameters equals to that of a previously processed job, it is directly inserted into the final queue Q_f, with the same timestamp and a sampling rate $s = 1$.

3. Jobs in Q_s are processed FIFO. They are inserted in the queue of the host Q_i where the corresponding dataset is stored, with the same timestamp. A fixed amount of time is assigned for the execution.

4. Every time a job leaves the scheduler and is inserted in one of the Q_i, a global execution plan is updated, that contains the busy times for every host in the system, obtained by the cost model associated to every execution.

5. The queues Q_i are populated by jobs with different sampling rates. A FIFO policy is applied to such queues. When a job is executed, if its sampling rate is less than 1, it will have an associated maximum execution time, assigned at step 3. After that time, if the sampling job has actually finished, it is inserted in the Q_f queue, with sampling rate $s = 1$, and timestamp given by the current time. If it has not finished, it is reinserted in the same Q_i with a smaller sampling rate s, timestamp set to the current time and a longer maximum execution time. This solution for scheduling jobs of unknown duration was originally proposed in [44].

6. Every time a job has finished we update the global execution plan.

7. When sampling is successfully finished, jobs are inserted in Q_f, where different possibilities are evaluated and the best option selected. Jobs in Q_f are processed in an SRFT fashion. Each job has an associated duration, obtained from the execution of sampling. Notice that sampling was preformed on the host where the dataset was

stored, while for the final execution we also consider the possibility of transferring the dataset to a more convenient location. Using (4.8) we are however able to predict job duration on a different machine.

All the procedure we have described relies on the fact that we can somehow control actual job duration, either with sampling, or with maximum allowed time for executions.

4.3.8 Evaluation

We designed a simulation framework to evaluate our MCT on-line scheduler, which exploits sampling as a technique for performance prediction. We thus compared our MCT+sampling approach with a *blind mapping strategy*. Since the blind strategy is unaware of actual execution costs, it can only try to minimize data transfer costs, and thus always maps the tasks on the machines that hold the corresponding input datasets. Moreover, it can not evaluate the profitability of parallel execution, so that sequential implementations are always preferred. Referring to the architectures for DDM systems proposed in Section 4.1.2, here we are comparing the performance of an architecture-driven scheduler (MCT+sampling) with those of a data-driven one (blind). As anticipated in Section 4.1.2, the simple data-driven model turns out to be less effective in scheduling both communications and computations of DDM on the K-Grid.

The simulated environment is similar to an actual Grid environment we have at disposal, and is composed of two clusters of three machines. Each cluster is interconnected by a switched fast Ethernet, while a slow WAN interconnection exists between the two clusters. The two clusters are homogeneous, but the machines of one cluster are *two times faster* than the machines of the other one. To fix simulation parameters, we actually measured average bandwidths b_{WAN} (100KB/s) and b_{LAN} (100MB/s) of the WAN and LAN interconnections, respectively.

We assumed that DM tasks to be scheduled arrive in a burst, according to an exponential distribution. They have random execution costs, but the x% of them corresponds to expensive tasks (1000 sec. as mean sequential execution time on the slowest machine), while the $(100 - x)$% of them are cheap tasks (50 sec. as mean sequential execution time on the slowest machine). Datasets D_i are all of medium size (50MB), and are randomly located on the machines belonging to the two clusters.

In these first simulation tests, we essentially checked the feasibility of our approach before actually implementing it within the K-Grid. Our goal was thus to evaluate mapping quality, in terms of *makespan*, of an optimal on-line MCT+sampling technique. We also assumed to also know in advance (through an oracle) the exact cost of the sampled tasks, instead of assuming an arbitrary small constant. In this way, since our MCT+sampling technique works in an optimal way, we can evaluate the maximal improvement of our technique over the blind scheduling one.

We analyzed the effectiveness of a centralized on-line mapper based on the MCT (Minimum Completion Time) heuristics [58, 87], which schedules DM tasks on a small organization of a K-Grid. The mapper does not consider node multitasking, is responsible for

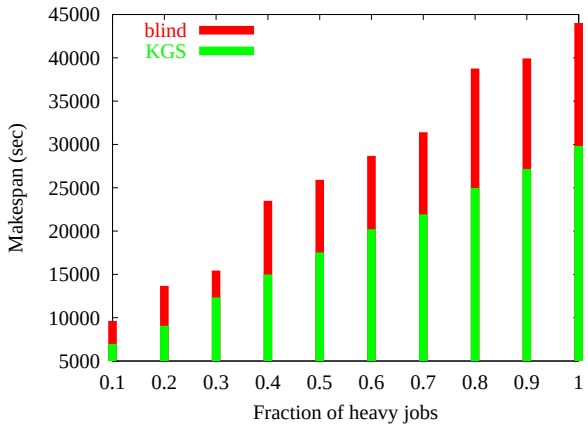


Figure 4.16: Comparison of makespans observed for different percentages of expensive tasks, when either a blind heuristics or our MCT+sampling one is adopted.

scheduling both dataset transfers and computations involved in the execution of a given task t_i, and also is informed about their completions. The MCT mapping heuristics adopted is very simple. Each time a task t_i is submitted, the mapper evaluates the *expected ready time* of each machine and communication links. The expected ready time is an estimate of the *ready time*, the earliest time a given resource is ready after the completion of the jobs previously assigned to it. On the basis of the expected ready times, our mapper evaluates all possible assignment of t_i, and chooses the one that reduces the completion time of the task. Note that such estimate is based on both estimated and actual execution times of all the tasks that have been assigned to the resource in the past. To update resource ready times, when data transfers or computations involved in the execution of t_i complete, a report is sent to the mapper.

Note that any MCT mapper can take correct scheduling decisions only if the expected execution time of a task is known. When no performance prediction is available for t_i, our mapper first generates and schedules $\widehat{t}_i$, i.e. the task t_i executed on the sampled dataset $\widehat{D}_i$. Unfortunately, the expected execution time of sampled task $\widehat{t}_i$ is unknown, so that the mapper has to assume that it is equal to a given small constant. Since our MCT mapper can not be able to optimize the assignment of $\widehat{t}_i$, it simply assigns $\widehat{t}_i$ to the machine that hosts the corresponding input dataset, so that no data transfers are involved in the execution of $\widehat{t}_i$. When $\widehat{t}_i$ completes, the mapper is informed about its execution time. On the basis of this knowledge, it can predict the performance of the actual task t_i, and optimize its subsequent mapping and scheduling.

Figures 4.17 illustrate two pairs of Gannt charts, which show the busy times of the six machines of our Grid testbed when tasks of different weights are submitted. In particular, each pair of charts is relative to two simulations, when either the blind or the

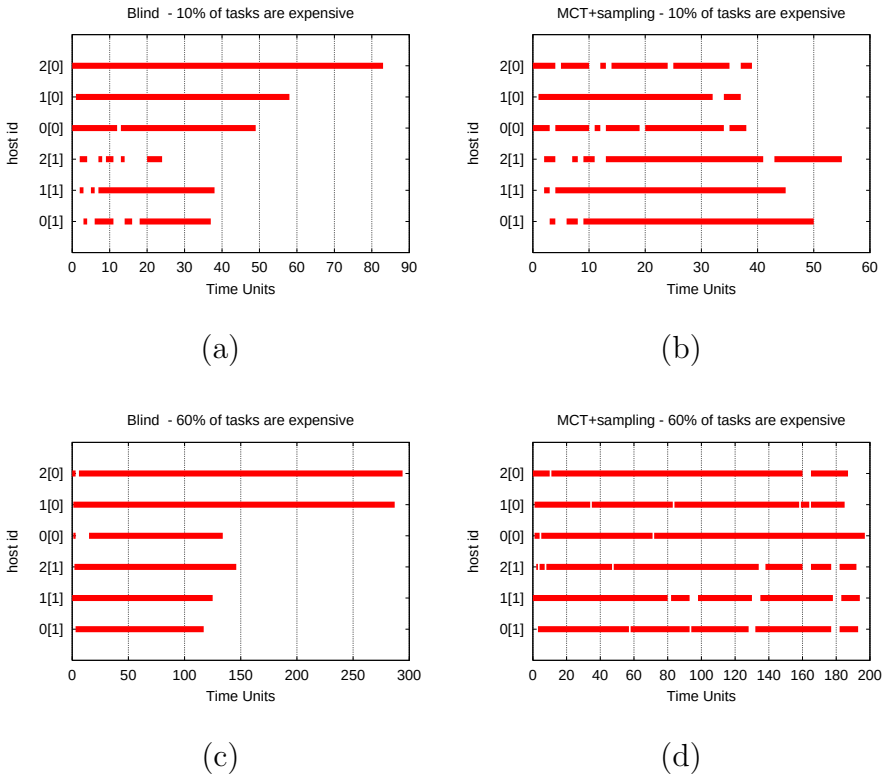


Figure 4.17: Gannt charts showing the busy times (in time units of 100 sec.) of our six machines when either the 10% (a,b) or the 60% (c,d) of the tasks are expensive: (a,b) blind scheduling heuristics, (c,d) MCT+sampling scheduling heuristics.

MCT+sampling strategy is adopted. Machine i of cluster j is indicated with the label i[j]. Note that when the blind scheduling strategy is adopted, since cluster 0 is slower than the other and no datasets are moved, the makespan on the slower machines results higher. Note that our MCT+sampling strategy sensibly outperforms the blind one, although it introduces higher computational costs due to the sampling process. Finally, Figure 4.16 shows the improvements in makespans obtained by our technique over the blind one when the percentage of heavy tasks is varied.

Chapter 5

Data Mining Template Library

In this Chapter we describe the Data Mining Template Library, a toolkit for the development of DM algorithms and applications, designed and implemented at the Rensselaer Polytechnic Institute, Troy, (NY) USA. The project is lead by professor M. J. Zaki. We took part to the design and first implementation of DMTL, during 2003 fall. We describe here the main ideas behind this projects and a preliminary implementation of the toolkit.

5.1 Introduction

The important results achieved in solving specific components of the complex process of extracting knowledge from massive datasets, often need to be harmonized into a more general framework that allows to fully exploit their potential. Since the Association Rule Mining (ARM) problem was first introduced, the cost of the most expensive phase of ARM has been reduced of orders of magnitude. At the same time, the notion of frequent pattern has been extended to other type of data like time ordered sequences, or trees and graphs. Also other kind of patterns are found to be more expressive for different application domains: so called *maximal* and *closed* patterns provide a more synthetic representation of interesting correlation inside data, that sometimes result to be more effective than traditional itemsets.

There is a need for tools and systems that bring together all such algorithms and techniques, allowing on one side for an easier deployment of applications based on such results, and on the other side for the development of new algorithms that could possibly improve current state of the art.

Our goal here is to develop a systematic solution to a whole class of common data mining tasks in massive, diverse, and complex datasets, rather than to focus on a specific problem. We are developing a prototype large-scale frequent pattern mining (FPM) toolkit, which is: i) Extensible and modular for ease of use and customization to needs of analysts, ii) Scalable and high-performance for rapid response on massive datasets.

The FPM toolkit is customizable to allow experts to build new, custom mining primitives and algorithms, as well as plug-and-play, following a generic programming paradigm [9]

for software development. The common generic core of FPM are built upon highly efficient "primitives" or building-blocks for: mining (search over complex high-dimensional spaces), pre-processing (sampling, discretization, feature selection, etc.), and post-processing (rule pruning, non-redundancy, interestingness, visualization).

The Data Mining Template Library (DMTL) forms the core of FPM toolkit. DMTL consists of a library of generic containers and algorithms, as well as persistency management for various FPM tasks. We describe the design and implementation of the FPM prototype for an important subset of FPM tasks, namely mining frequent itemsets and sequences. Our main contributions are as follows:

- The design and implementation of generic data structures and algorithms to handle itemset and sequence patterns.
- Design and Implementation of generic data mining algorithms for FPM, such as depth-first and breadth-first search.
- DMTL's persistent/out-of-core structures for supporting efficient pattern frequency/statistics computations using a tightly coupled database management systems (DBMS) approach.
- Native support for both a vertical and horizontal database format for highly efficient data mining.
- DMTL's support for pre-processing steps like data mapping and discretization of continuous attributes and creation of taxonomies. etc.

5.2 Related Work

The current practice in frequent pattern mining basically falls into the paradigm of incremental algorithm improvement and solutions to very specific problems. While there exist tools like MLC++ [53], which provides a collection of algorithms for classification, and Weka [99], which is a general purpose java library of different data mining algorithms including itemset mining, classification and clustering, as well as other commercial products, there is no unifying theme or framework, there is little database support, scalability is questionable, and there is little support for KDD process. Moreover, these tools are not designed for handling complex pattern types like trees and graphs.

Previous research in integrating mining and databases has mainly looked at SQL support. DMQL [41] is a mining query language to support common mining tasks. MSQL [45] is an extension of SQL to generate and selectively retrieve sets of rules from a large database. The MINE RULE SQL operator [62] and Query flocks [95] extend the semantics of association rules, allowing more generalized queries to be performed. A comprehensive study of several architectural alternatives for database and mining integration were studied in [83]. This body of work is complementary to ours, since these SQL operators can be

used as a front end to the FPM toolkit. Also, our toolkit is optimized for the class of frequent patterns.

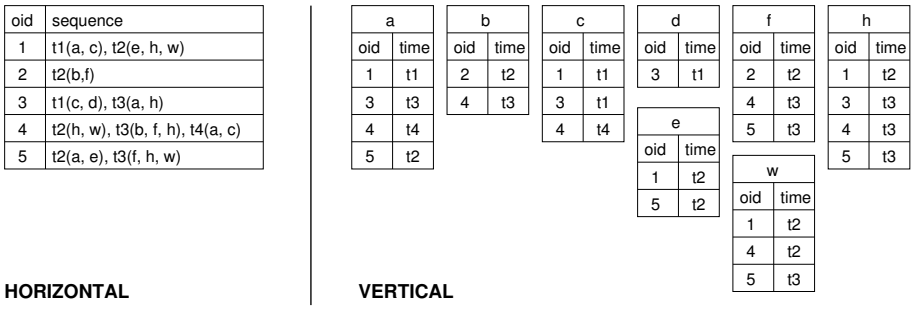


Figure 5.1: Data Formats for Sequences: horizontal (left) and vertical (right)

5.3 Problem definition

We defined in Section 2.1 the problem of finding frequent itemsets in a database, where itemsets are collections of items which on turn are a rather generic representation for almost any kind of data type. The notion of pattern can be generalized even further in order to cover other kinds of patterns besides itemsets, like sequences or trees. The input of an FPM algorithm is a generic database D where a collection of objects $D = \{o_1, o_2, ...o_n\}$ is stored. Each object o_i is a collection of elements from a finite set I. If the database represents sales data from a supermarket, I represents the set of items sold in the supermarket. If the database represents a relational table then the set I represents all the attribute-value pairs in the table. Finally, if the database represents a web access log file, then the objects are navigation sessions and elements in the object are timestamped sequences of html pages. More complex data structure can be represented in D, like trees or graphs.

5.3.1 Database Format

Most of the previous FPM work has utilized horizontal database format for storing the objects. However, as we mentioned in Section 2.4, vertical formats are in general more efficient for itemset mining [69, 103, 86, 23, 106, 47, 18], sequence mining [104, 10, 70] and tree mining [108]. In a vertical database each label is associated with its corresponding oid list, the set of all oids (object identifiers) where it appears, along with additional pattern specific information, like timestamp for sequences. Oid lists are therefore an extension of tidlists for itemset mining. As in our algorithm DCI, bitvectors can be used to have a more compact representation of oid lists [69, 86, 18, 10].

Similarly to the case of itemset transactions (see Figure 2.1), Figure 5.1 contrasts the horizontal and vertical data formats for sequence mining. In the horizontal approach, each object has an oid along with the sequence of itemsets comprising the object. Thus object with $oid = 1$ is the sequence (a, c) at time t_1 and (e, h, w) at time t_2 . In contrast, the

vertical format maintains for each label (and itemset) its oid list, a set of all oids where it occurs and the corresponding timestamps. For example, label a appears in oids 1, 3, 4, and 5.

Mining algorithms using the vertical format have shown to be very effective and usually outperform horizontal approaches [103, 86, 104, 10, 108]. The reason is that the vertical format narrows down the data accessed to only the relevant subset, allows one to mine frequent patterns via specialized intersection operations on the vertical oid-lists, and supports a variety of search strategies. For example, for itemset mining, to find if $\{a, d\}$ is frequent we intersect oid lists of a and d. As already commented, the drawback of this approach is related to the number of intersections that is necessary to perform to calculate the support of long itemsets. Specific optimizations are needed in order to reduce the number of intersections. On the other hand, for all those methods relying on intersection operations, such optimizations turn out to be effective, thus allowing for performance gain in a whole class of algorithms.

When short patterns are mined, the horizontal format allows to achieve better performance. In fact, during the first steps of FPM algorithms oid lists can still be too long to be performed in core. For this reason DCI adopts a hybrid approach, horizontal on the first steps and vertical as soon as the pruned vertical dataset fits into main memory.

Our DMTL implementation thus provides *native* database support for both horizontal and vertical object formats.

5.4 Containers and Algorithms

Following the ideology of generic programming, DMTL provides a standardized, general, and efficient implementation of frequent pattern mining tasks by isolating the concept of data structures or containers, as they are called in generic programming, from algorithms. DMTL provides container classes for representing different patterns (such as itemsets and sequences) and collection of patterns, containers for database objects (horizontal and vertical), and containers for temporary mining results (such as new oid-lists produced via intersections). These container classes support persistency when required. Functions specific to the containers are included as member functions. Generic algorithms, on the other hand are independent of the container and can be applied on any valid container. These include algorithms for performing intersections of the vertical oid-lists for itemsets or sequences or other patterns. Generic algorithms are also provided for mining itemsets and sequences (such as Apriori/GSP [4, 91], Eclat/SPADE [103, 104]), as well as for finding the maximal or closed patterns in a group of patterns. Finally DMTL provides support for the database management functionality, pre-processing support for mapping data in different formats to DMTL's native formats, as well as for data transformation (such as discretization of continuous values).

In this section we focus on the containers and algorithms for mining. In later sections we discuss the database support in DMTL as well as support for pre-processing and post-processing.

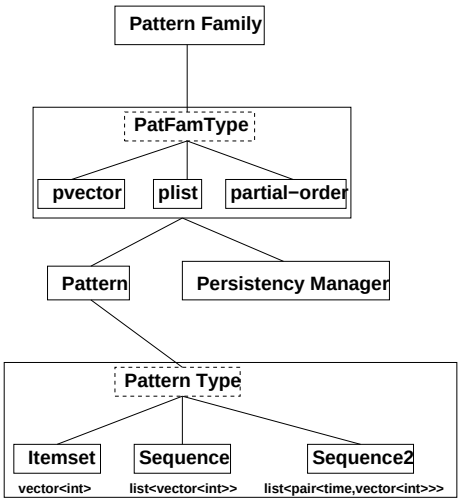


Figure 5.2: DMTL Container Hierarchy

5.4.1 Containers: Pattern and PatternFamily

Figure 5.2 shows the different DMTL container classes and the relationship among them. At the lowest level set the different kinds of pattern-types one might be interested in mining (such as itemsets, sequences, and several variants). A pattern uses the base pattern-type classes to provide a generic container. There are several pattern family types (such as pvector, plist, etc.) which together with a persistency manager class make up different pattern family classes. More details on each class appears below.

Pattern. In DMTL a pattern is a generic container, which can be instantiated as an itemset, sequence, tree or a graph, specified by means of a template argument called pattern-type (P); A snippet of the generic pattern class is shown in Figure 5.3.

A generic pattern is simply a pattern-type whose frequency we need to determine in a larger collection or database of patterns of the same type. A pattern container has as data members the pattern-type and support, and functions like `serialize_read`, and `serialize_write` (to read/write the pattern from/to some stream). It also contains operations to check for equality (==, !=) and to check if another pattern is a sub-pattern (`includes()`). All DMTL classes use the dmtl namespace to avoid naming conflicts with other libraries (we will omit the dmtl namespace from now on, it is assumed by default).

Pattern Type. This allows users to select the type of pattern they want to mine, and as long as certain operations are defined on the pattern-type all the generic algorithms provided by DMTL can be used. A snippet of the Itemset pattern-type is shown in Figure 5.4.

Typically, an itemset is represented by pattern-type `vector<int>`, denoting a set of items (`int` in this case), The pattern-type class must in addition provide certain operations required by the generic Pattern class. The operations in the Pattern class are all defined

```
namespace dmtl{
  //Pattern-type P
  template <class P> class Pattern{
  public:
    void set_support(int sup);
    int get_support();
    P & get_pattern();
    void set_pattern(P & p);
    //Read/Write pattern from/to a stream
    void serialize_read(istream & S);
    void serialize_write(ostream & S);
    //Check if p is sub-pattern
    bool includes(Pattern<P> &p);
    //Pattern equality, assignment
    bool operator== (Pattern<P> &p);
    bool operator!= (Pattern<P> &p);
    Pattern<P> & operator= (Pattern<P> &p);
  private:
    int support;
    P pattern;
  }
} //namespace dmtl
```

Figure 5.3: Generic Pattern Class

in terms of the corresponding operations in the specific pattern-type class. For instance the `includes()` function of Pattern may be defined as follows:

```
   //Check if p is sub-pattern
bool includes(Pattern<P> &p){
return pattern.includes(p.get_pattern());
}
```

The sequence pattern-type is defined as `list<vector<int>>`, denoting an ordered list of itemsets. This is achieved by defining the pattern-type as follows:

```
   class Sequence{
typedef typename
list<vector<int>> pattern-type;
//Other functions
}
```

The rest of the function prototypes remains the same as in Itemset class, but a sequence pattern-type must provide specific implementations of the different operations.

```
class Itemset{
public:
  //Definition of Itemset
  typedef typename vector<int> pattern_type;
  //Is p a sub-pattern?
  bool includes (pattern_type & p);
  bool operator== (pattern_type &p);
  bool operator!= (pattern_type &p);
  pattern_type & operator= (pattern_type &p);
private:
  pattern_type P; //The itemset
}
```

Figure 5.4: Itemset Pattern Type

For instance the `includes()` function will be defined differently for itemset and sequence pattern-types. The main source of flexibility is that developers can easily define new types of patterns to suit their needs and once the operations are defined on them all the generic algorithms of DMTL can be used on the new pattern types. For instance suppose we want to include a time field along with the different itemsets in a sequence, we can define a new sequence type as follows:

```
   class Sequence2{
typedef typename
list<pair<time, vector<int>>>
pattern-type;
//Other functions
}
```

Here we have a sequence that is a list of (`time,vector<int>`) pairs, where time is a user-defined type to note when each event occurs.

Pattern Family. In addition to the basic pattern classes, most FPM algorithms operate on a collection of patterns. The generic class `PatternFamily` provides methods that manipulate a group of patterns of the same type. A snippet of the pattern family class is shown here:

```
//Pattern Family Type PatFamType, Database DB
template <class PatFamType> class PatternFamily{

public:
  //Type of Pattern Family
  typedef typename PatFamType pat_fam_type;
  //Type of Pattern
```

```
  typedef typename PatFamType::pattern pattern;
  typedef typename PatFamType::iterator iterator;
                                                                    10
  //Add/Remove pattern to/from family
  void add(pattern & p);
  void remove(pattern &p);
  //Read/Write patterns from/to stream
  void serialize_read(stream & S);
  void serialize_write (stream & S);
  //Find maximal/closed patterns in family
  PatternFamily & maximal();
  PatternFamily & closed();
  // Find all subsets of p                                          20
  PatternFamily & lower_shadow(pattern & p);
  //Find all supersets of p
  PatternFamily & upper_shadow(pattern & p);
  //Count support of all patterns
  void count(DB &db);

private:
  PatFamType PatFam; //Pattern Family
}
```

The pattern family is a generic container to store groups of patterns, specified by the template parameter `PatFamType`. `PatFamType` represents some persistent class provided by DMTL, that provides seamless access to the members, whether they be in memory or on disk. All access to patterns of a family is through the iterator provided by the `PatFamType` class. `PatternFamily` provides generic operations to add and remove patterns to/from the family, to serialize the whole family to/from a stream (such as a file/screen), to find the maximal or closed patterns in the family, and to find the set of all matching super-patterns (called upper-shadow) and sub-patterns (called lower-shadow) of a given pattern, within the family. Finally, the `count()` function finds the support of all patterns, in the database, using functions provided by the DB class.

Pattern Family Type. DMTL provides several persistent classes to store groups of patterns. Each such class is templatized on the pattern-type and a persistency manager class PM. An example is the `pvector` class shown below:

```
   //Pattern P, Persistency Manager PM
template <class P, class PM>
class pvector{
typedef typename P pattern;
typedef typename PM persistency-manager;
//Functions for persistency
}
```

`pvector` is a persistent vector class. It has the same semantics as a STL vector with added memory management and persistency. Thus a pattern family for itemsets can be defined in terms of the pvector class as follows: `PatternFamily<pvector<Itemset, PM>>`. Another class is `plist<P,PM>`.

5.4.2 Persistency and Database Class

An important aspect of DMTL is to provide a user-specified level of persistency for all DMTL classes. To support large-scale data mining, DMTL provides automatic support for out-of-core computations, i.e., memory buffer management, via the persistency manager class PM. The PatternFamilyType class uses the persistency manager (PM) to support the buffer management for patterns. The details of implementation are hidden from PatternFamily; all generic algorithms continue to work regardless of whether the family is (partially) in memory or on disk. The implementation of a persistent container (like pvector) is similar to the implementation of a volatile container (like STL vector); the difference being that instead of pointers one has to use offsets and instead of allocating memory using `new` one has to request it from the persistency manager class. More details on the persistency manager will be given later.

We saw above that PatternFamily uses the `count()` function to find the support of all patterns in the family, in the database; at the end of `count()` all patterns have their support field set to their frequency in the database. DMTL provides *native* support for both the horizontal and vertical database formats. The generic `count()` algorithm does not impose any restriction on the type of database used, i.e., whether it is horizontal or vertical. The `count()` function uses the interface provided by the DB class, passed as a parameter to `count()`, to get pattern supports. More details on the DB class and its functions will be given later.

5.4.3 Generic FPM Algorithms

The FPM task can be viewed as a search over the pattern space looking for those patterns that match the minimum support constraint. As we said in Chapter 2, in itemset mining, the search space is the set of all possible subsets of I. If $|I| = m$ the search space can be as big as 2^m. For sequence mining, the search space is the set of all possible sub-sequences formed from I. For m items, there are $O(m^k)$ sequences of length up to k [104]. Various search strategies are possible leading to several popular variants of the mining algorithms. DMTL provides generic algorithms encapsulating these search strategies; by their definition these algorithms can work on any type of pattern: Itemset, Sequence, Tree or Graph.

In the intermediate steps of mining most algorithms several *candidate* patterns are evaluated, and only the ones that satisfy the support threshold are kept for further processing. FPM algorithms can differ in the way new candidates are generated. The dominant approach is that of *complete* search, whereby we are guaranteed to generate and test all frequent patterns. Note that completeness doesn't mean exhaustive, since we can use pruning to eliminate useless branches in the search space. *Heuristic* generation sacrifices

completeness for the sake of speed. At each step, it only examines a limited number of "good" branches. *Random* search to locate the maximal frequent patterns is also possible. Typical methods that can be used here include genetic algorithms, simulated annealing, and so on, but these methods have not received much attention in FPM literature, since a lot of emphasis has been placed on completeness. Likewise, DMTL provides several complete search schemes over the pattern space.

Breadth-first and Depth-first Search. Most approaches have used a *breadth-first* (BFS, also called bottom-up or level-wise) search to enumerate the frequent patterns. This approach is adopted for example by the DCI algorithm (see Section 2.4). It starts with the single elements in I, and counts their support. It then finds those that are frequent, denoted F_1, and constructs candidate patterns of size 2, denoted C_2 by extending the size 1 patterns by adding a new element from I. All patterns in C_2 are counted in the database, and the frequent ones are found, denoted by F_2. At any given intermediate stage k, candidate patterns of size k, C_k, are constructed by extending the frequent patterns of size $k-1$ by one more element from I, and these candidates are counted in the database to yield the frequent patterns of size k, denoted F_k. The process stops when no new frequent patterns are found.

Recent approaches have used the *depth-first* (DFS) enumeration scheme. Here instead of counting the entire level C_k at a time, only a select subset of candidate patterns are counted and extended an element at a time until no extensions are possible. After that one returns to the remaining groups of patterns within level k. The process repeats until all extensions of every single element in I have been counted. DMTL provides support for both breadth-first and depth-first search for frequent patterns.

Grouping Patterns: Equivalence Classes. While performing BFS or DFS it is beneficial to first order the patterns into groups, a common strategy being grouping by the common prefix, giving rise to *prefix-based* equivalence classes [103, 104]. The idea then is to mine over classes, rather than individual patterns, i.e., a class represents elements that the prefix can be extended with to obtain a new frequent pattern, and no extension of an infrequent prefix has to be examined. DMTL supports such grouping through the means of a generic group iterator class, that puts all patterns satisfying a given condition (e.g., having same prefix), into the same class.

Mining Algorithms There are two main steps for mining frequent patterns: to generate candidates to be counted and to count the support in the database. DMTL provides several generic algorithms for mining all patterns. One approach uses BFS to explore the search space. The algorithm is completely generic in that it can work for any pattern provided certain conditions are met. For instance, a snippet of the code might look like:

```
template <class PatFamType>
void BFS-Mine (PatternFamily<PatFamType> &pf,
DB &db, ...)  {
```

```
typedef typename
PatFamType::pattern-type P;
typedef typename
PatFamType::persistency-manager PM;
....
}
```

The generic BFS mining algorithm takes in a pattern family and the database. The types of patterns and persistency manager are specified by the pattern family type. The BFS algorithm in turn relies on other generic subroutines for creating groupings, for generating candidates, and for support counting. For instance different grouping functions can be defined in the PatternFamily class, e.g., prefix group. The prefix grouping class will use other methods provided by the pattern/pattern-type class to check if two patterns have the same prefix and if so will group them into the same class. The BFS algorithm also needs to extend candidate patterns by adding another element. Once again generic subroutines provided by the pattern family can be used to extend the patterns to create new candidates. They in turn will rely on the method provided by pattern/pattern-type class to extend a given pattern. When `BFS-Mine` is invoked the user must supply the pattern family type, the database and other parameters like the grouping strategy and extension strategy. For counting a candidate collection the functions provided by the database class can be used. The generic DFS mining algorithm, `DFS-Mine`, is similar to `BFS-Mine` in all respects except that it makes a recursive call to itself to process the patterns in a DFS manner.

The generic BFS-Mine algorithm, when used in conjunction with a horizontal database and no grouping is exactly the same as the classic Apriori algorithm [4] for Itemset patterns, and is the same as GSP [91] for Sequence patterns. When BFS/DFS-Mine are used with a vertical database and prefix-based grouping, they give rise to Eclat [103] for itemsets and SPADE [104] for sequences.

5.5 Persistency/Database Support

In this section we discuss the database and persistency support provided in DMTL. As we mentioned earlier DMTL provides native database support for both the horizontal and vertical data formats. It is also worth noting that since in many cases the database contains the same kind of objects as the patterns to be extracted (i.e., the database can be viewed as a pattern family), the same database functionality used for horizontal format can be used for providing persistency for pattern families.

It is relatively straightforward to store a horizontal format object, and by extension, a family of such patterns, in any object-relational database. Thus the persistency manager for pattern families can handle both the original database and the patterns that are generated while mining. DMTL provides the required buffer management so that the algorithms continue to work regardless of whether the database/patterns are in memory or on disk.

```
//vat-type V
template <class V> class VAT{

public:
  typedef typename V vat_type;
  //add/remove entry to/from \vat\
  void add (vat_type & vbt);
  void remove (vat_type & vbt);
private:
  vector<vat_type> vat_body;
}
```

Figure 5.5: VAT class

More details will be discussed later.

To provide native database support for objects in the vertical format, DMTL adopts a fine grained data model, where records are stored as *Vertical Attribute Tables* (VATs). Given a database of objects, where each object is characterized by a set of properties or attributes, a VAT is essentially the collection of objects that share the same values for the attributes. For example, for a relational table, `cars`, with the two attributes, `color` and `brand`, a VAT for the property `color=red` stores all the transaction identifiers of cars whose color is red. The main advantage of VATs is that they allow for optimizations of query intensive applications like data mining where only a subset of the attributes need to be processed during each query. As was mentioned earlier these kinds of vertical representations have proved to be useful in many data mining tasks [103, 104, 108]. We next introduce the VAT data model focusing on itemset and sequences; extending the model to handle patterns like trees and graphs is part of future work.

5.5.1 Vertical Attribute Tables

We consider a pattern to be composed of a collection of attribute-value (AV) pairs, with additional constraints as necessary. For instance an itemset is simply a set of AV pairs (e.g., {`color=red, brand=ford`} could be an itemset of length 2). A sequence on the other hand specifies a temporal ordering on the AV pairs.

In DMTL there is typically one VAT per pattern. A VAT is an entity composed of a *body*, which contains the list of object identifiers in which a given pattern occurs. For storing database sequences a VAT needs, in addition, a `time` field for each occurrence of the pattern. VATs are first created per unique AV pair (patterns of length 1) and later on in the mining, per itemset/sequence pattern made up of a collection of AV pairs. The class definition of a VAT is shown in Figure 5.5.

Depending on the pattern type being mined the vat-type class may be different. For instance for itemset mining it suffices to keep only the object identifiers where a given

itemset appears. In this case the vat-type is simply an `int` (assuming that oid is an integer). On the other hand for sequence mining one needs not only the oid, but also the time stamp for the last AV pair in the sequence. For sequences the vat-type is then `pair<int, time>`, i.e., a pair of an `int`, denoting the oid, a nd `time`, denoting the time-stamp. Different vat-types must also provide operations like equality testing (for itemsets and sequences), and less-than testing (for sequences; a oid-time pair is less then another if they have the same oid, and the first one happens before the second).

Given the generic setup of a VAT, DMTL defines another generic algorithm to join/intersect two VATs. For instance in vertical itemset mining, the support for an itemset is found by intersection the VATs of its lexicographic first two subsets. A generic intersection operation utilizes the equality operation defined on the vat-type to find the intersection of any two VATs. On the other hand in vertical sequence mining the support of a new candidate sequence is found by a temporal join on the VATs, which in turn uses the less-than operator defined by the vat-type. Since the itemset vat-type typically will not provide a less-than operator, if the DMTL developer tries to use temporal intersection on itemset vat-type it will generate a *compile time* error! This kind of concept-checking support provided by DMTL is extremely useful in catching library misuses at compile-time rather than at run-time.

DMTL provides support for creating VATs during the mining process, i.e., during algorithms execution, as well as support for updating VATs (add and delete operations). In DMTL VATs can be either persistent or non-persistent. Finally DMTL uses indexes for a collection of VATs for efficient retrieval based on a given attribute-value, or a given pattern.

VAT Creation

We consider the following three basic data models for itemset and sequence mining: relational tables, transactions and sequences. Many FPM algorithms run on these three types of databases. We show how VATs can be defined for each of them and effectively used for the execution of FPM algorithms.

Relational Tables to VATs In a relational table columns are different attributes, each one with its own domain of possible values. Rows are objects with given values for each attribute. For each AV pair `(attribute=A, value=V)`, we have a VAT which holds the collection of object identifiers where `A=V`. Figure 5.6 shows an example of the VATs obtained from a relational table.

It is clear that in general the number of VATs associated with a table will be as big as the number of all possible combinations `(attribute, value)`. Nevertheless, information stored in the collection of VATs corresponding to a given relational table is the same (there is no redundancy).

Transactions to VATs Transactions are variable length subsets of elements/items taken from a given set I. They can also be represented as rows in a relational table where columns

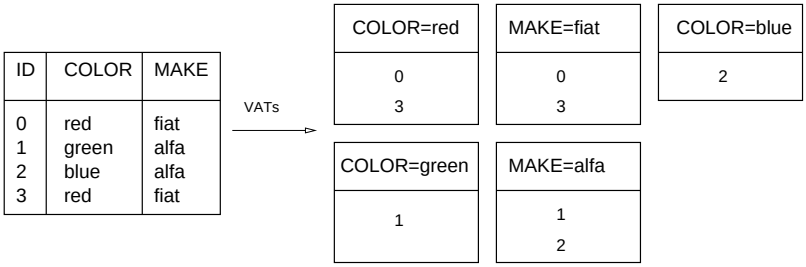


Figure 5.6: Mapping relations onto VATs.

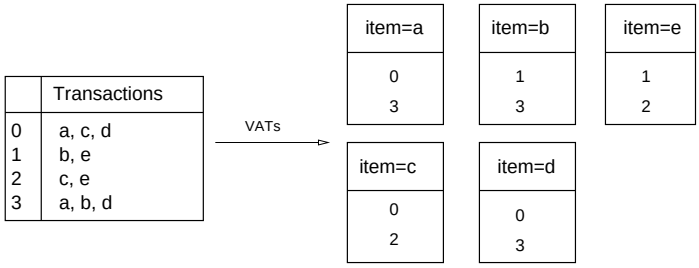


Figure 5.7: Mapping transactions onto VATs.

are items in I and each column contains a 1 (item is present in the transaction) or 0 (item is not present in the transaction). In this case, a VAT stores the set of object identifiers (transaction ids) where each item appears. Figure 5.7 shows the mapping from transaction to VATs, which corresponds to the vertical representation shown earlier in Figure 5.1.

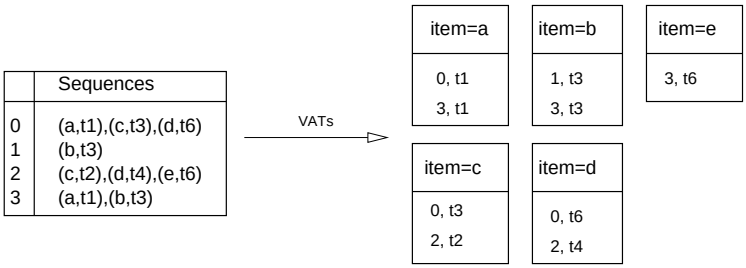


Figure 5.8: Mapping sequences onto VATs.

Sequences to VATs Sequences are transactions, called sessions, where each item, called an event, is associated with a timestamp that tells when the event happened. In this case we still have a VAT for each possible event, but for every object identifier we also store the corresponding timestamp, as shown in Figure 5.8.

It is important to observe that at this level we do not impose any restriction on the data type of attributes, items, or events. They can very generally be integers, categories or floating point real numbers.

5.5.2 Database/Persistency-Manager Classes

The database support for VATs and for the horizontal family of patterns is provided by DMTL in terms of the following classes, which are illustrated in Figure 5.9.

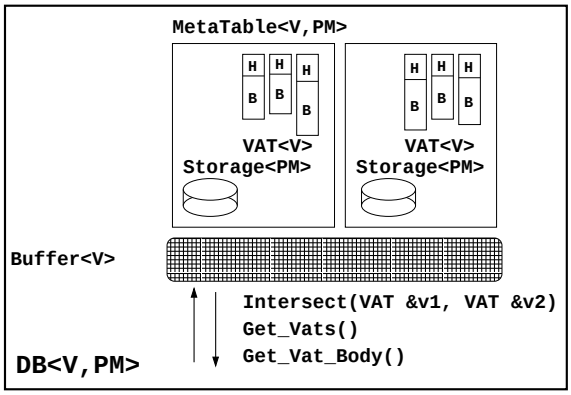


Figure 5.9: High level overview of the different classes used for Persistency

`Vat-type`: A class describing the vat-type that composes the body of a VAT, for instance `int` for itemsets and `pair<int,time>` for sequences.

`VAT<class V>`: The class that represents VATs. This class is composed of a collection of records of vat-type V.

`Storage<class PM>`: The generic persistency-manager class that implements the physical persistency for VATs and other classes. The class PM provides the actual implementations of the generic operations required by Storage. For example, PM_metakit and PM_gigabase are two actual implementations of the `Storage` class in terms of different DBMS like Metakit [98], a persistent C++ library that natively supports the vertical format, and Gigabase [52], an object-relational database.

`MetaTable<class V, class PM>`: This class represents a collection of VATs. It stores a list of VAT pointers and the adequate data structures to handle efficient search for a specific VAT in the collection. It also provides physical storage for VATs. It is templatized on the vat-type V and on the `Storage` implementation PM.

`DB<class V, class PM>`: The database class which holds a collection of `Metatables`. This is the main user interface to VATs and constitutes the database class DB referred to in previous sections. It supports VAT operations such as intersection, as well as the operations for data import and export. The double template follows the same format as that of the `Metatable` class.

`Buffer<class V>`: A fixed-size main-memory buffer to which VATs are written and from which VATs are accessed, used for buffer management to provide seamless support for main-memory and out-of-core VATs (of type V).

As previously stated, the DB class is the main DMTL interface to VATs and the persistency manager for patterns. It has as data members an object of type Buffer<V> and a collection of MetaTables<V,PM>.

The Buffer<V> class is composed of a fixed size buffer which will contain as many VAT bodies. When a VAT body is requested from the DB class, the buffer is searched first. If the body is not already present there, it is retrieved from disk, by accessing the Metatable containing the requested VAT. If there is not enough space to store the new VAT in the buffer, the buffer manager will (transparently) replace an existing VAT with the new one. A similar interface is used to provide access to patterns in a persistent family or the horizontal database.

The MetaTable class stores all the pointers to the different VAT objects. It provides the mapping between the patterns, called header, and their VATs, called the body, via a hashed based indexing scheme. In the figure the H refers to a pattern and B its corresponding VAT. The Storage class provides for efficient lookup of a particular VAT object given the header.

VAT Persistency VATs can be in one of three possible states of persistence:

- volatile: the VAT is fully loaded and available in main memory only.
- buffered: the VAT is handled as if it were in main memory, but it is actually kept on disk in an out-of-core fashion.
- persistent: the VAT is disk resident and can be retrieved after the program execution, i.e.: the VAT is inserted in the VAT database.

Volatile VATs are created and handled by directly accessing the VAT class members. Buffered VATs are managed from the DB class through Buffer functions. Buffered VATs must be inserted into the file associated with a Metatable, but when a buffered VAT is no longer needed, its space on disk can be freed. A method for removing a VAT from disk is provided in the DB class. If such method is not called, then the VAT will be persistent, i.e., it will remain in the metatable and in the storage associated with it after execution.

Buffer Management The Buffer class provides methods to access and to manage a fixed size buffer where the most recently used VATs/patterns are stored for fast retrieval. The idea behind the buffer management implemented in the Buffer class is as follows.

A fixed size buffer is available as a linear block of memory of objects of type V (Figure 5.10). Records are inserted and retrieved from the buffer as linear chunks of memory. To start, the buffer is empty. When a new object is inserted, some data structures are initialized in order to keep track of where every object is placed so it can be accessed later. Objects are inserted one after the other in a round-robin fashion. When there is no more space left in the buffer, the least recently used (LRU) block (corresponding to one entire VAT body, or a pattern) is removed. While the current implementation provides a LRU buffering strategy, as part of future work we will consider more sophisticated buffer replacement strategies that closely tie with the mining.

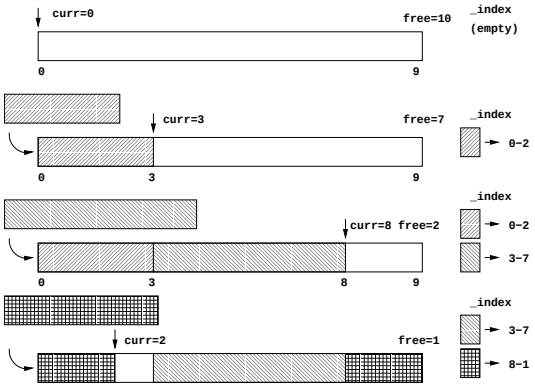


Figure 5.10: Buffer Management: The buffer manager keeps track of the current position in the buffer (curr), the amount of free space (free), and an index to keeps track of where an object is. As new objects are inserted these are updated and when an existing object is replaced when the buffer becomes full.

Storage Physical storage of VATs and pattern families can be implemented using different storage systems, such as a DBMS or ad-hoc libraries. In order to abstract the details of the actual system used, all storage-related operations are provided in a generic class, `Storage`. Implementations of the `Storage` class for MetaKit [98] and Gigabase [52] backends are provided in the DMTL. Other implementations can easily be added as long as they provide the required functionality.

The `DB` class is a doubly templated class where both the vat-type and the storage implementation need to be specified. An example instantiation of a `DB` class for itemset patterns would therefore be `DB<int,PM_metakit>` or `DB<int,PM_gigabase>`.

5.6 Pre-processing Support

We described in the previous sections how traditional data models - namely tables, transactions and sequences - can be mapped to VATs. Now we show how the actual data are encoded into VATs. Since our data can be of any type (discrete, continuous, categorical or even of more complex types), we use an internal representation of VATs for efficient mining. Moreover, the same source data can be treated differently according to the particular analysis we might want to perform. For example, starting from the same database of, say, continuous values, we might want to run some FPM algorithms that relies on one discretization of the continuous values, but after that we may want to run another algorithm for a different discretization.

DMTL provides support for dynamic mapping of data into VATs at run-time over the same base database using a *mapping* class, which transforms original attribute values into *mapped values* according to a set of user specified mapping directives, contained in a configuration file. For every input database there has to be an XML configuration file. For the definition of the syntax of such file, we follow the approach presented in [60] by Talia

et al.. The format of such file is the following.

```
<?xml version="1.0"?>
<!DOCTYPE Datasource SYSTEM "dmtl_config.dtd">
<Data model=relational source=ascii_file>
<Access> [...]  </Access>
<Structure>
<Format> [...]  </Format>
<Attributes> [...]  </Attributes>
</Structure>
</Data>
```

Attributes Configuration used for mapping attribute values are contained in the `<Structure>` section. The `<Format>` section contains the characters used as record separator and field separator. An `<Attribute>` section must be present for each attribute (or column) in the input database. Such section might be something like: `<Attribute name="price" type="continuous" units="Euro" ignore="yes"> [ ... ] </Attribute>`. Possible attributes for the `<Attribute>` tag are: `name`: the name of the attribute, `type`: one of continuous, discrete, categorical, `units`: the unit of measure for values (currency, weight, etc.), `ignore`: should a VAT be created for this attribute or not.

Mapping The mapping information is enclosed in the `<Mapping>` section. Mapping can be different for categorical, continuous or discrete fields. For continuous values we can specify a fixed step discretization within a range:

```
<Attribute name="price" type="continuous">
<Mapping min="1.0" max="5.0" step=".5">0
</Mapping> </Attribute>
```

In this case the field `price` will be mapped to `(max-min)/step = (5-1)/.5 = 10` values, labeled with integers starting from `0`. It is also possible to specify non-uniform discretizations, omitting the `step` attribute and explicitly specifying all the ranges and labels:

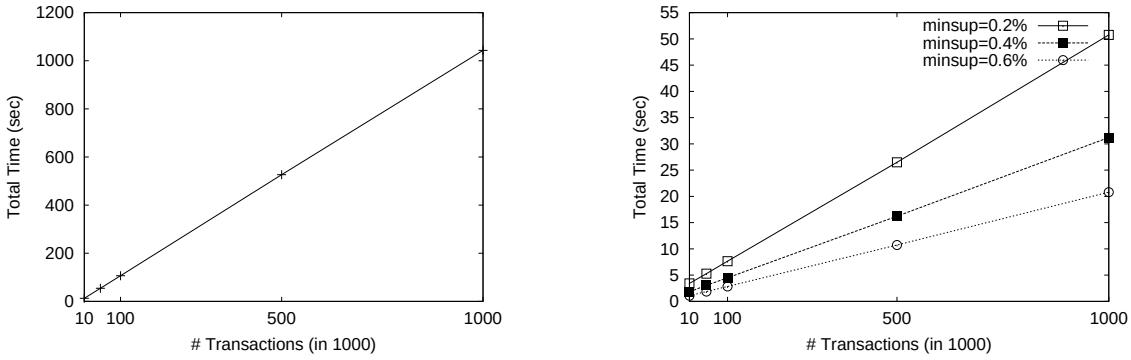
```
<Attribute name="price" type="continuous">
<Mapping min="0.5" max="2.0">1</Mapping>
<Mapping min="2.0" max="20.0">2</Mapping>
<Mapping min="20.0" max="100.0">3</Mapping>
<Mapping min="100.0" max="-1">4</Mapping>
</Attribute>
```

For categorical values we can also specify a mapping, that allows for taxonomies or other groupings. Below we are classifying items into higher order categories. For example, turkey and chicken are both items of type meat.

```
<Attribute name="item" type="categorical"
units="food_category" ignore="yes">
<Mapping val="corona">beer</Mapping>
<Mapping val="adams">beer</Mapping>
<Mapping val="chianti">wine</Mapping>
<Mapping val="turkey">meat</Mapping>
<Mapping val="chicken">meat</Mapping>
</Attribute>
```

5.7 Experiments

DMTL is implemented using C++ Standard Template Library [9]. We present some experimental results on the time taken by DMTL to load a transactional database and to perform itemset mining on it. We used the IBM synthetic database generator [4], to create datasets (T10I4DxK) with transactions ranging from x=10K to x=1000K (or 1 million).



(a) (b)

Figure 5.11: Loading (a) and mining (b) times.

Figure 5.11 (a) shows the time taken to convert the transactional data into VATs, and Figure 5.11 (b) shows the mining time for several values of the minimum support s. We see that DMTL scales linearly with the number of transactions. While the time to convert to the VAT format and to load it into `PM_metakit` incurs some overhead, the mining time of DMTL is as good as a flat file run.

5.8 Future work

We conclude this Chapter with a brief analysis of the possible extensions to the current implementation of DMTL.

5.8.1 Implementation of DCI into DMTL

In order to extend DMTL current set of built-in algorithms, we plan to insert our DCI algorithm for FSC into the toolkit. Since DCI is based essentially on *Apriori*, with a set of highly optimized strategies to be adopted according to dataset or hardware characteristics, we believe that including DCI ideas into the toolkit will result in a more general benefit, also on other components.

Currently the work of porting DCI is still in its early stage. We review here the main points of this operation.

- First of all, DCI relies on a vertical bitmap based representation of the dataset. DMTL already provides a native vertical dataset representation, based on VATs, but at the moment VAT bodies, corresponding to tidlists, are implemented as `vectors`. It would therefore be necessary to have a bitmap based implementation of VATs, where the body of a VAT is a linear and contiguous chunk of memory. Using bitvectors can lead to important performance improvements, since intersections can be performed more efficiently using word level `and` operations. The impact of this on the current version of DMTL should only be a different implementation of the VAT class.

- Another benefit coming from the use of bitvector VATs is that, if all length-k VATs are allocated together, they can be stored in contiguous memory regions, thereby allowing for high locality in their access which on turn would allow processor caching strategies to be effective. Since itemsets are accessed according to their lexicographic order, high locality is indeed exhibited in the process of itemsets enumeration. As a first step, it would be enough to store all length-1 VATs in a contiguous buffer. In this way we should obtain something similar to the bitmap representation of the dataset used by the original implementation of DCI, as presented in Chapter 2.

- The multiple optimization strategies adopted by DCI, rely on the ability of skipping sections of 0's in the tidlists (per sparse datasets) and identical sections of tidlists (in dense datasets). Moreover, for sparse dataset DCI adopts a pruning strategy according to which empty columns (i.e. transactions) are skipped during intersections. In DMTL this would reflect in the development of methods in the `database` class that can tell whether a dataset is dense or sparse. This can be achieved with an inspection of the bitmap representation of the dataset. Once this knowledge is available, the optimizations for sparse and dense datasets could be implemented in the intersection operations. Intersections could, as it is now the case in DCI, *remember* which are the tidlists (i.e. bitmap VAT bodies) sections to be skipped.

- A more effective optimization at the intersection level is the use of a cache for storing the partial results of a $k-$way intersection. We have demonstrated (Section 2.4.4) how a small cache of size $k-2$ times the size of a tidlist is enough to dramatically reduce the number of actual intersections performed, due to the lexicographic order with which candidates are enumerated.

5.8.2 DMTL extensions

A set of possible extensions to DMTL, among others, can be enumerated as follows:

- FPM might involve more articluated constraints than the simple minimum support threshold. For this reason the minimum support condition should be abstracted from the several code framents where it is currently replicated and performed in a separate method that could possibly implement other constraint as well.
- The XML configuration file could be easily extended to include sampling. In the `<format>...</format>` section, for example, a `<sampling>` element could be introduced, with proper attributes that specify the sampling rate and the sampling strategy. Multiple sampling strategy could in fact be available.
- Although currently DMTL is designed mainly for a sequential hardware platform, its general architecture allows for an easy extension. One possible architectural extention could be achieved providing a distributed loosely coupled implementation of the persistency manager class, using a client-server DBMS like Mysql or others. Such implementation could be useful for the development of parallel and distributed FPM algorithms.

Chapter 6

Conclusions

The development in storage, network and computing technologies in last decade, has brought us today a potentially endless amount of data. According to R. Grossman [35] "Today a 10 GB data set can easily be produced with a simple mistake, 100 GB - 1 TB data sets are common, and data sets larger than a Terabyte are becoming common". On the other hand, the corresponding technology aimed at extracting valuable and usable knowledge from such data, not always has undergone a similar exponential development. To state it with J. Han's words "we are data rich but information poor" [42]. Filling such knowledge-gap is the challenge that data miners are currently called to face.

Many research topics take part in the active discipline of DM. From new algorithms to knowledge modeling, through the application of DM techniques to new and diverse application domains. Common to all different fields of DM is the limitation posed by performance issues. Input datasets can be, and in general are, huge so that algorithms are required to scale reasonably to those dimensions. Datasets can be distributed. DM applications must therefore be able to run in a similar environment, taking into account all the consequent problems (how to move data? how to consider privacy constraints?). Data have limited lifetime and change continuously. DM applications therefore must be able to incrementally update the knowledge extracted from such data in an online fashion. Finally, computations are heavy and their costs difficult to model. DM systems have to take advantage of the aggregate resources to modern HPC architectures, applying smart policies for resource management.

In our work we have focused on the performance of Data Mining. Starting from the analysis of a common DM task, namely Association Rule Mining (ARM), we have designed and implemented an algorithm for the most computational intensive part of ARM, i.e. Frequent Set Counting. The Direct Count and Intersect (DCI) algorithm puts together many optimizations proposed separately in an original way, achieving good performance results with respect of state of the art algorithms either sequential or parallel. Such promising results motivate further research in the field of efficient FSC. In particular we plan to include more optimization into the DCI algorithm and increase the degree of adaptivity to the dataset specific features (like density or length of patterns). Also, a wider class of patterns are to be included in DCI, like sequences or trees and graphs.

Streaming data pose the problem of incremental mining algorithms and of maintaining updated the knowledge extracted along a period of time that exceeds the data production rate. We studied a problem of Web Usage Mining (WUM) and designed a WUM system (SUGGEST) that is able to personalize the pages of a web server, based on the analysis of users behavior in past navigation sessions. SUGGEST, implemented as a module of Apache, is tightly coupled with the web server and is able to deliver significant personalized content with a limited overhead on the server ordinary performance. SUGGEST builds its knowledge from an incremental graph partitioning algorithm that maintains clusters of *related pages*, built according on users navigation patterns. One limitation of SUGGEST regards its ability to handle dynamic web pages, which are becoming dominant in the web scenario. We plan to extend SUGGEST functioning to such pages as well. Moreover, we are currently working on the utilization of SUGGEST on production web servers in order to test its actual performance on real life applications.

Large scale distributed systems are often the natural environment for DM applications. It is actually here that the "move data" nightmare comes into play: could ever be a feasible option that of moving huge volumes of data across geographic networks? Many arguments have been addressed at pushing this option aside. We showed by means of a simulation that what might sound as an unrealistic resort is in fact a viable option, in the case where many DM computations and data transfers take place. We therefore studied the high level architecture of a scheduler for distributed DM tasks that have to be executed on large scale distributed architectures, like the Knowledge Grid. We devised a heuristic scheduling policy and an original cost model for DM tasks, based on sampling as a method for performance prediction. The results obtained from the simulation need to be extended and tested on real Knowledge Grids.

Specific results found in solving single DM problems can be generalized and applied to a wider class of algorithms and applications. For example most of the ideas in FSC algorithms can effectively be applied to general Frequent Pattern Mining. The use of vertical dataset representation and tidlist intersection as a method for support counting, instead of horizontal database scan, is an approach that has already been shown to be effective in itemset, sequence and tree mining. Similarly, access to data can be abstracted form the physical details of accessing records on disk, encoding them in the most suitable format for a specific DM task and possibly performing preprocessing like discretization or taxonomy. High level support can be developed for the design of DM applications that hide such details to users and generalize previous results. During fall 2002, we joined a project for the design of a Data Mining Template Library, at the Rensselaer Polytechnic Institute of Troy, NY, USA. under the supervision of prof. M. J. Zaki. The goal of the project is to develop a systematic solution to a whole class of common data mining tasks in massive, diverse, and complex datasets. The first step toward this goal is the development of a prototype large-scale frequent pattern mining (FPM) toolkit, which is: i) Extensible and modular for ease of use and customization to needs of analysts, ii) Scalable and high-performance for rapid response on massive datasets.

The body of work presented in this thesis covers an apparently wide range of problems. As we said several times, they are all concerned with the performance of DM systems. We

would like to conclude this thesis with some general ideas that in our view and on the basis of the work presented here, should be kept in mind for obtaining performance out of DM systems:

Data adaptivity DM systems must have the ability to adapt themselves to the input data peculiarities. No unique solution has ever proved to be the *best* one, in any DM sub-domain. Efficient DM systems must provide multiple strategy solutions to all DM problems faced with suitable heuristics to dynamically chose the most effective strategy.

Resource adaptivity Efficient DM algorithms should always provide the ability of managing different available computing resources. In our view DM algorithms should natively provide support for disk resident data structures, transparent access to distributed resources and scalability both in time and space.

Generalization Patterns and the techniques to mine them should be as general as possible in order to ensure that performance results are not limited to specific algorithms or single kernels.

Performance Awareness Suitable cost models should be available to DM applications, together with the control of the accuracy of the result found by the application. This would allow for the possibility of *trading* performance with accuracy directly inside the application.

Bibliography

[1] D. Abramson, J. Giddy, I. Foster, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *International Parallel and Distributed Processing Symposium Cancun, Mexico*, 2000.

[2] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing*, 2000. Special Issue on High Performance Data Mining.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association between Sets of Items in Massive Databases. In *ACM-SIGMOD 1993 Int'l Conf. on Management of Data*, pages 207–216. ACM, 1993. Washington D.C. USA.

[4] R. Agrawal, H. Manilla, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast Discovery of Association Rules in Large Databases. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transaction On Knowledge And Data Engineering*, 8:962–969, 1996.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 439–450. ACM Press, May 2000.

[7] Jain A.K. and Dubes R.C. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[8] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. Gridftp protocol specification. Technical report, GGF GridFTP Working Group Document, 2002.

[9] M. H. Austern. *Generic Programming and the STL*, volume 3. Addison Wesley Longman, Inc., May-June 2000.

[10] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *SIGKDD Int l Conf. on Knowledge Discovery and Data Mining*, July 2002.

[11] Peter Bailey, Nick Craswell, and David Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Information Processing & Management*, (to appear).

[12] R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego. Implementation issues in the design of I/O intensive data mining applications on clusters of workstations. In *Proc. of the 3rd Work. on High Performance Data Mining, (IPDPS-2000), Cancun, Mexico*, pages 350–357. LNCS 1800 Spinger-Verlag, 2000.

[13] R. Baraglia and P. Palmerini. Suggest: A web usage mining system. In *Proceedings of IEEE International Conference on Information Technology: Coding and Computing*, 2002.

[14] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *ACM SIGKDD Explorations Newsletter*, 2(2):66–75, December 2000.

[15] Francine Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[16] J. P. Bradford and J. Fortes. Performance and memory access characterization of data mining applications. In *Proceedings of Workshop on Workload Characterization: Methodology and Case Studies*, 1998.

[17] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 255–264, New York, May 1997. ACM Press.

[18] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: a maximal frequent itemset algorithm for transactional databases. In *In Intl. Conf. Data Engineering*, Apr. 2001.

[19] M. Cannataro, C. Mastroianni, D. Talia, and Trunfio P. Evaluating and enhancing the use of the gridftp protocol for efficient data transfer on the grid. In *Proc. of the 10th Euro PVM/MPI Users' Group Conference*, 2003.

[20] M. Cannataro, D. Talia, and P. Trunfio. Design of distributed data mining applications on the knowledge grid. In *Proceedings NSF Workshop on Next Generation Data Mining*, pages 191–195, November 2002.

[21] Bin Chen, Peter Haas, and Peter Scheuermann. A new two-phase sampling based algorithm for discovering association rules. In *Proceedings ACM-SIGKDD Conference*, Edmonton, Canada, July 2002.

[22] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: towards an architecture for the distributed management and analysis of large scientific datasets. *J. of Network and Comp. Appl.*, (23):187–200, 2001.

[23] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of the 15th ICDE Int. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, 1999. IEEE Computer Society.

[24] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors : Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[25] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Handbook of Algorithms and Theory of Computation*, chapter Dynamic Graph Algorithms, Chapter 22. CRC Press, 1997.

[26] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1998.

[27] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[28] I. Foster and C. Kasselman. *The Grid: blueprint for a future infrastructure*. Morgan Kaufman, 1999.

[29] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 2(11):115–128, 1997.

[30] Enrique Frias-Martinez and Vijay Karamcheti. Reduction of user perceived latency for a dynamic and personalized site using web-mining techniques. In *Proceedings of WebKDD*, pages 47–57, 2003.

[31] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.

[32] F. Geerts, B. Goethals, and J. Van den Bussche. A tight upper bound on the number of candidate patterns. In N. Cercone, T.Y. Lin, and X. Wu, editors, *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 155–162. IEEE Computer Society, 2001.

[33] Bart Goethals. *Efficient Frequent Itemset Mining*. PhD thesis, Limburg University, Belgium, 2003.

[34] R. Grossman and A. Turinsky. A framework for finding distributed data mining strategies that are intermediate between centralized strategies and in-place strategies. In *KDD Workshop on Distributed Data Mining*, 2000.

[35] R. L. Grossman and R. Hollebeek. *Handbook of Massive Data Sets*, chapter The National Scalable Cluster Project: Three Lessons about High Performance Data Mining and Data Intensive Computing. Kluwer Academic Publishers, 2002.

[36] Robert Grossman, Stuart Bailey, Balinder Mali Ramau, and Andrei Turinsky. The preliminary design of papyrus: A system for high performance, distributed data mining over clusters. In *Advances in Distributed and Parallel Knowledge Discovery*. AAAI/MIT Press, 2000.

[37] Robert L. Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and Raju R. Namburu, editors. *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, 2001.

[38] The Data Mining Group. PMML 2.0. `http://www.dmg.org/pmmlspecs_v2/pmml_v2_0.html`.

[39] Y. Gu, M. Hong, X. Mazzucco, and R. L. Grossman. Sabul: A high performance data transfer protocol. *submitted to IEEE Communications Letters*, 2003.

[40] E. H. Han, G. Karypis, and Kumar V. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):337–352, May/June 2000.

[41] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql: A data mining query language for relational databases. In *SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1999.

[42] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[43] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, USA, 2000.

[44] Mor Harchol-Balter. Task assignment with unknown duration. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, 2000.

[45] T. Imielinski and A. Virmani. Msql: A query language for database mining. *Data Mining and Knowledge Discovery: An International Journal,*, (3):373–408, 1999.

[46] Yarsun Hsu Jin-Soo Kim, Xiaohan Qin. Memory characterization of a parallel data mining workload. In *Proceedings of Workshop on Workload Characterization: Methodology and Case Studies*, 1998.

[47] Gouda K. and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE Int l Conf. on Data Mining*, Nov. 2001.

[48] *Proceedings of the First SIAM Workshop on Data Mining and Counter Terrorism*, 2002.

[49] H. Kargupta, W. Huang, S. Krishnamrthy, B. Park, and S. Wang. Collective principal component analysis from distributed, heterogeneous data. In *Proc. of the principals of data mining and knowledge discovery*, 2000.

[50] H. Kargupta, W. S. K. Huang, and E. Johnson. Distributed clustering using collective principal components analysis. *Knowledge and Information Systems Journal*, 2001.

[51] H. Kargupta, B. Park, E. Johnson, E. Sanseverino, L. Silvestre, and D. Hershberger. Collective data mining from distributed vertically partitioned feature space. In *Proc. of Workshop on distributed data mining, International Conference on Knowledge Discovery and Data Mining*, 1998.

[52] Konstantin Knizhnik. Gigabase, object-relational database management system. http://sourceforge.net/projects/gigabase.

[53] R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using MLC++, a machine learning library in C++. *International Journal of Artificial Intelligence Tools*, 4(6):537–566, 1997.

[54] S. Krishnaswamy, S.W. Loke, and A. Zaslavsky. Cost models for distributed data mining. In *12th Int. Conference on Software Engeneering and Knowledge Engeneering*, 2000.

[55] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[56] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *Lecture Notes in Computer Science*, 1880:36, 2000.

[57] J. Liu, Y. Pan, K. Wang, and J. Han. Mining Frequent Item Sets by Opportunistic Projection. In *Proc. 2002 Int. Conf. on Knowledge Discovery in Databases (KDD'02), Edmonton, Canada*, 2002.

[58] M. Maheswaran, A. Shoukat, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th HCW*, 1999.

[59] M. Marzolla and P. Palmerini. Simulation of a grid scheduler for data mining. Esame per il corso di dottorato in informativa, Universita' Ca' Foscari, Venezia, 2002.

[60] C. Mastroianni, D. Talia, and P. Trunfio. Managing heterogeneous resources in data mining applications on grids using xml-based metadata. In *Proceedings of The 12th Heterogeneous Computing Workshop*, 2002.

[61] Jesus Mena. *Data Mining Your Website*. Digital Press, United States of America, 1999. http://www.bh.com/digitalpress.

[62] R. Meo, C. Psaila, and S. Ceri. A new sql-like operator for mining association rules. In *Proc. of 22nd Intl. Conf. Very Large Databases*, 1996.

[63] B. Mobasher, R. Cooley, and J. Srivastava. Automatic personalization based on web usage mining. *Communications of the ACM*, 43(8):142–151, august 2000.

[64] Bamshad Mobasher, Namit Jain, Eui-Hong (Sam) Han, and Jaideep Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical Report TR 96-050, University of Minnesota, 1996.

[65] Miki Nakagawa and Bamshad Mobasher. A hybrid web personalization model based on site connectivity. In *Proceedings of WebKDD*, pages 59–70, 2003.

[66] Olfa Nasraoui and Christopher Petenes. Combining web usage mining and fuzzy inference for website personalization. In *Proceedings of WebKDD*, pages 37–46, 2003.

[67] S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of the 3rd Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK 2001, LNCS 2114*, pages 71–82, Munich, Germany, 2001. LNCS Spinger-Verlag.

[68] S. Orlando, P. Palmerini, and R. Perego. On Statistical Properties of Transactional Datasets. In *Proc. of 19th ACM Symposium on Applied Computing, SAC*, 2003.

[69] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proceedings of IEEE International Conference on Data Mining*, 2002.

[70] S. Orlando, R. Perego, and C. Silvestri. An new algorithm for gap constrained sequence mining. In *Proc. of 19th ACM Symposium on Applied Computing, SAC*, 2003.

[71] B. Park and H. Kargupta. *Data Mining Handbook*, chapter Distributed Data Mining: Algorithms Systems and Applications. (to be published), 2004.

[72] J. S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 1995.

[73] C. L. Parkinson and R. Greenstonen, editors. *EOS Data Products Handbook*. NASA Goddard Space Flight Center, 2000.

[74] S. Parthasarathy. Towards network-aware data mining. In *Workshop on Parallel and Distributed Data Mining, held along with IPDPS01*, 2001.

[75] Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *Winter Simulation Conference*, pages 1037–1044, 1997.

[76] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. 2001 Int. Conf. on Data Mining (ICDM'01)*, November 2001.

[77] Mike Perkowitz and Oren Etzioni. Adaptive web sites: Conceptual cluster mining. In *International Joint Conference on Artificial Intelligence*, pages 264–269, 1999.

[78] C. Pizzuti and D. Talia. P-autoclass: Scalable parallel clustering for mining large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):629–641, 2003.

[79] A. L. Prodromidis, P. K. Chan, and S. J. Stolfo. Meta-learning in distributed data mining systems: Issues and approaches. In *Advances in Distributed and Parallel Knowledge Discovery*. AAAI/MIT Press, 2000.

[80] F. J. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *Knowledge Discovery and Data Mining*, pages 23–32, 1999.

[81] N. Ramakrishnan and A. Y. Grama. Data Mining: From Serendipity to Science. *IEEE Computer*, 32(8):34–37, 1999.

[82] G. Ramesh, W. A. Maniatty, and M. J. Zaki. Feasible itemset distributions in data mining: Theory and application. In *22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 2003.

[83] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternatives and implications. In *In ACM SIGMOD Intl. Conf. Management of Data*, 1998.

[84] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21th VLDB Conf.*, pages 432–444, Zurich, Switzerland, 1995.

[85] Jennifer M. Schopf. A general architecture for scheduling on the grid. Technical Report ANL/MCS-P1000-10002, Argonne National Laboratory ANL/MCS-P1000-10002., 2002.

[86] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 2000.

[87] H. J. Siegel and Shoukat Ali. Techniques for Mapping Tasks to Machines in Heterogeneous Computing Systems. *Journal of Systems Architecture*, (46):627–639, 2000.

[88] H. J. Siegel and A. Shoukat. Techniques for mapping tasks to machines in heterogeneous computing systems. *Journal of Systems Architecture*, 2000.

[89] David Skillicorn. The case for datacentric grids. Technical Report 451, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6, November 2001.

[90] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[91] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.

[92] K. Stoffel and A. Belkoniene. Parallel k-means clustering for large datasets. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685. Springer-Verlag, 1999.

[93] D. Talia and M. Cannataro. Knowledge grid: An architecture for distributed knowledge discovery. *Communications of the ACM*, 2002.

[94] Robert Thau. Design considerations for the Apache Server API. *Computer Networks and ISDN Systems*, 28(7–11):1113–1122, 1996.

[95] D. Tsur, J. Ullman, S. Abitboul, C. Clifton, R. Motwani, and S. Nestorov. Query flocks: A generalization of association rule mining. In *Proc. of ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[96] Sudharshan Vazkhudai, Jennifer Schopf, and Ian Foster. Predicting the performance of wide area data transfers. In *proc. of IPDPS*, 2002.

[97] Dan Verton. Congressman says data mining could have prevented 9-11. *Computerworld*, 2002.

[98] Jean-Claude Wippler. Metakit. http://www.equi4.com/metakit/.

[99] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[100] Kun-lung Wu, Philip S. Yu, and Allen Ballman. Speedtracer: A web usage mining and analysis tool. *IBM Systems Journal*, 37(1), 1998.

[101] Tak Woon Yan, Matthew Jacobsen, Hector Garcia-Molina, and Dayal Umeshwar. From user access patterns to dynamic hypertext linking. *Fifth International World Wide Web Conference*, May 1996.

[102] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining*, 7(4):14–25, December 1999.

[103] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, May/June 2000.

[104] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 1/2(42):31–60, Jan-Feb 2001.

[105] M. J. Zaki and K. Gouda. Fast Vertical Mining Using Diffsets. In *9th Int. Conf. on Knowledge Discovery and Data Mining, Washington, DC*, 2003.

[106] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data Mining*, Apr. 2002.

[107] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. In *7th Int. Workshop on Research Issues in Data Engineering (RIDE)*, pages 42–50, Birmingham, UK, 1997.

[108] Mohammed J. Zaki. Efficiently mining trees in a forest. In *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (KDD)*, July 2002.

[109] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proc. of KDD-2001*, 2001.