High Performance Closed Frequent Itemsets Mining inspired by Emerging Computer Architectures

Claudio Lucchese

Supervisor
prof. Salvatore Orlando

PhD Coordinator
prof.ssa Annalisa Bossi

January 2008
Author’s Web Page:  http://hpc.isti.cnr.it/~claudio

Author’s e-mail:  clucches@dsi.unive.it

Author’s address:

Dipartimento di Informatica
Università Ca’ Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it
Abstract

This thesis is devoted to the design of high performance algorithms for the extraction of closed frequent itemsets from transactional databases. Our interest in high performance algorithms is inspired by emerging computer architectures.

We believe that high performance not only means to design fast algorithms but also to find new solutions to challenging problems. We also use our experience in constrained pattern mining to support this claim. We discuss in detail some of the issues related to frequent pattern mining algorithms: computational complexity, data size and the opportunities provided by emerging computer architectures. For each of the three aforementioned issues, we contribute a novel algorithm.

DCI-CLOSED is a new algorithm for mining closed frequent itemsets. This is the only algorithm that can mine efficiently dense datasets without maintaining the running collection of already discovered closed itemsets. OOC-CLOSED is the first algorithm for mining closed frequent itemsets in secondary memory. Finally, we propose MT-CLOSED: the first parallel algorithm for mining closed frequent itemsets.

The present trend suggests that emerging computer architectures will become more complex over time, with tenths or hundreds of CPUs, complex cache hierarchies, and probably streaming and pipelined frameworks. With this work, we believe to provide important contributions in the direction of harnessing modern computer architectures and, at the same time, to give new insights on the mining of closed patterns.
# Contents

1 Introduction .................................................. 1
  1.1 What is Data Mining? .................................. 1  
  1.2 Web search as a data mining process .............. 4  
  1.3 Challenges in data mining ............................ 6  
  1.4 Contribution of this thesis .......................... 7  

2 The importance of high performance in frequent itemsets mining  9
  2.1 Introduction to frequent itemsets mining ........... 9  
  2.2 Frequent itemsets mining algorithms ............... 10  
  2.3 Issue #1: computational cost ....................... 13  
  2.4 Issue #2: data size ................................. 16  
  2.5 Issue #3: parallel mining and emerging computer architectures .. 18  
  2.6 Other issues ......................................... 19  
  2.7 Conclusions .......................................... 21  

3 CONQUEST: constraint-based pattern discovery  23
  3.1 The constrained pattern mining paradigm ............ 23  
  3.2 A review of constraint-based algorithms ............ 25  
  3.3 A new data-reduction framework for constrained pattern mining ..................... 28  
    3.3.1 The basic mining engine ....................... 30  
    3.3.2 The CONQUEST algorithm ...................... 31  
  3.4 Experimental evaluation ............................ 33  
  3.5 The CONQUEST system ................................ 39  
  3.6 Conclusions .......................................... 41  

4 DCI-CLOSED: a new algorithm for mining closed frequent itemsets  45
  4.1 Motivations for closed frequent itemsets .......... 45  
  4.2 The closed frequent itemsets mining problem ...... 46  
  4.3 The state-of-the-art of closed frequent itemsets mining ..................... 48  
    4.3.1 A level-wise algorithm ....................... 48  
    4.3.2 First generation depth-first algorithms ...... 49  
    4.3.3 A second generation of algorithms .......... 54  
  4.4 The DCI-CLOSED algorithm ......................... 55  
    4.4.1 Detecting order-preserving generators ....... 56  
    4.4.2 Computing closures ......................... 58  
    4.4.3 Mining dense datasets ....................... 58  
  4.5 Performance analysis .............................. 64
## Contents

4.5.1 Performance comparisons ..................................... 64  
4.5.2 Time efficiency of duplicate check ............................. 70  
4.5.3 Space efficiency of duplicate check ............................. 73  
4.5.4 Scalability .......................................................... 76  
4.6 Conclusions ............................................................. 77  

5 OOC-CLOSED: the first out-of-core CFIM algorithm ................. 79  
5.1 Introduction ............................................................ 79  
5.2 Partitioning strategies ................................................. 80  
  5.2.1 Partitioning the input dataset .................................... 81  
  5.2.2 Partitioning the search space .................................... 82  
5.3 Spurious itemsets detection in search space partitioning approaches ............................................. 84  
5.4 The mining engine: DCI-CLOSED ................................... 87  
5.5 Creating Partitions ..................................................... 87  
5.6 OOC-CLOSED: an new algorithm for mining closed frequent itemsets from secondary memory ........................................... 89  
5.7 Experimental evaluation .............................................. 89  
5.8 Conclusions ............................................................. 93  

6 Harnessing modern computer architectures with MT-CLOSED ........ 95  
6.1 Introduction ............................................................ 95  
6.2 Parallel pattern mining on modern computer architectures .......... 97  
  6.2.1 First generation algorithms ...................................... 97  
  6.2.2 Second generation algorithms .................................... 98  
  6.2.3 Further considerations on CFIM algorithms ....................... 99  
6.3 The MT-CLOSED algorithm ........................................... 100  
  6.3.1 Load partitioning and scheduling ............................... 102  
  6.3.2 First-level dataset projection .................................... 104  
6.4 Experimental evaluation .............................................. 105  
  6.4.1 Scalability .......................................................... 105  
  6.4.2 SIMD instruction set ............................................. 110  
6.5 Conclusions ............................................................. 112  

Conclusions ................................................................. 113  

A Description of datasets .................................................. 115  
  A.1 Description of the datasets used in this thesis .................... 115  

Bibliography ................................................................. 119
Introduction

1.1 What is Data Mining?

Today’s information society has become, with no doubt, a restless producer of data of many kinds. During the early stages, the rapid growth of the information technology has set ourselves to be targets of an overwhelming flow of information; advertisements are a clear example of this. Nowadays, we are all conscious or unconscious data generators, and most of the results of this creation process is stored in huge and evolving databases.

A diverse amount of data is being collected, from traditional supermarket transactions, credit card records, phone calls records, census statistics, and GPS tracks to other less usual kinds of data such as astronomical images, molecular structures, DNA sequences and medical records.

Of course, it did not take a long time to understand that these records are an extremely valuable source of information. Profiling users in order to understand their tastes and to take advantage of their behaviour is a primary business goal. Also, an increasing amount of efforts are spent to understand DNA sequences.

The will and the need to benefit from this largely available data, was the fertile land were the discipline of data mining was grown.

In this sense, data mining is a natural evolution of the information science and technology [30][31]. Data mining is a relatively young discipline (the first international conference on Knowledge Discovery and Data Mining was held in 1995), having its roots mainly in statistics and in other strongly related fields such as artificial intelligence, machine learning, pattern recognition and database technology. The border between one discipline and another is not sharp, and for this reason it is not easy to give a good definition of data mining. Among the many, I did choose only two of them, the first of them being the historical one:

Data mining is “the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data”.

Fayyad et al. [19]
The extracted information must be interesting, novel and useful. An important actor is now treading the stage: the user. It is up to him to judge whether the information extracted is useful or not. In the data mining perspective, there is no objective function (e.g. statistical significance) able to tell about the validity of a model. Rather, a complicated interaction with the user is required to discover the fragment of information that is subjectively interesting.

Data mining is only one part of what is referred to as Knowledge Discovery in Databases (KDD), that is the complex process able to convert raw data into useful knowledge about the data (see Fig. 1.1).

Starting from the raw data, the process of data cleaning and integration takes place. Data coming from different sources is filtered from noise and inconsistencies and possibly re-organized in a single data warehouse. Then, only interesting portions of data are retrieved. This includes noise removal, non-trivial normalization, feature selection, dimensionality reduction and other similar techniques. The resulting data has to be transformed in a proper format, suitable to be mined. The knowledge finally extracted is presented to the analyst which is in charge of judging about its interestingness and eventually tuning the parameters of the previous steps and restarting from the beginning. This iterative and interactive process will end once user's objectives are fulfilled.

Thus, data mining is not a once-and-for-all activity. The same data collection maybe analyzed an unlimited number of times. Also, new objectives may become interesting, and different kinds of information may be sought in different ways as the analyst's focus changes over time.

Typically, there are four kinds of tasks a data mining system can accomplish, or, said in other words, there are four different materializations of the knowledge extracted. These are predictive modeling, cluster analysis, anomaly detection and
1.1. What is Data Mining?

Association analysis (see Fig. 1.2).

Predictive modeling aims at building a model that can predict a target variable as a function of other explanatory ones. We distinguish between classification and regression respectively when the target variable is discrete or continuous. Detecting whether a patient has a particular disease based on his medical history is a classification task, while predicting tomorrow’s temperature based on today’s weather measurements is a regression task.

Cluster analysis is able to describe a dataset as a set of classes such that objects belonging to the same class are similar (intra-cluster similarity) and objects in different clusters are not (inter-cluster dissimilarity). In this case objects are not labeled, typically because labels are not known in advance. Clustering can be applied to find groups of similar customers.

Anomaly detection consists in discovering objects that significantly differs from all the other observations in the data. Typical applications are detection of bank frauds and network attacks.

Lastly, association analysis is used to discover strongly correlated patterns or sets of events, usually with the form of implication rules. Market basket analysis is one of this kind, where an interesting rule may be “who buys X always buys Y and Z”. Another application is the one of finding groups of genes having related functionalities.

Association analysis is actually a new kind of data modeling born thanks to data mining research community. While, for the other tasks, there is a lot in common...
with the fields of statistics and machine learning. However, remember that data mining is *driven by the data*. The peculiarities of this data raised the need for new non-traditional analysis techniques \[33, 34\]. The data mining discipline designed new techniques able to handle (i.e. to build models out of) large amounts of data, accomplishing goals that could not be fulfilled with traditional methods. Data makes the difference, and here is where the second definition of data mining will help us:

“Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner.”.

Hand et al.\[35\]

In statistics, data is collected in order to answer a specific question. Statisticians are thus concerned with “primary data analysis”. In fact, wide sub-disciplines of statistics try to understand what is the best way to collect data. When it comes to data mining, we deal with “secondary data analysis” since no activity takes place during the collection process. Usually, large amounts of data are explored with different purposes than the ones it was gathered for. These are sometimes called “opportunity” or “convenience” samples, as opposed to the classical “random” samples, and this is why they are named “observational data”. In a sense, this convenient data weakens traditional data analysis. In addition, it has lots of noise, missing values, contaminations etc. Data quality is a very important issue dealt during the first step of the KDD process.

Moreover, typical datasets are gigabytes or terabytes large, or even more. In the next section we will use web search as an illustrative example where data happens to be very large, but such large databases occur in all real life problems. This is not only an issue for what regards consolidating, storing and accessing data. Worse than that, data may happen to be distributed or evolving over time. Data mining tries to overcome the difficulties arising in these kinds of scenarios. This is one reason why, as opposed to statistics where models are very important, data mining has from the beginning stressed the importance of algorithms and their design\[34\].

Finally, when considering the problem of finding interesting patterns in association analysis or in outlier detection, traditional techniques would assess the statistical significance of each pattern. Since exploiting a large number of tests does not protect against the detection of spurious relationships, those techniques do not apply in the case of such large data collections.

In conclusion, we can say that data mining brought us novel contributions thanks to its strong roots in traditional disciplines.

1.2 Web search as a data mining process

The web, or better, search engines’ job, is a very fashionable example of a knowledge discovery process. We use the entire web search process as a toy example, even if
1.2. Web search as a data mining process

it is the subject of several mature scientific disciplines. However, this example will help to fix what has been said before and to introduce a few challenges in data mining that are described later.

The web is not only about text search. In fact, many issues and interesting fields of research emerged very recently. Their seed has been the large availability of data, and in particular the so called Web 2.0. Disregarding the various definition of this new shape of the web, what we really care is the amount of minable content generated by users. Links between pages are generated by the user, but also emails, query logs, browsing sessions and so on are a valuable source of information. The rule is: the more data the better. The reason why this user authored information is important can be explained if we look at it as a form of additional human knowledge embedded in the data. In a sense it can be considered a materialization of both users’ background knowledge and users’ will.

If we think about a simple web search (see Fig. 1.3), then the first step is of course crawling. Web pages have to be gathered all in one place, at least according to a traditional approach. These are physically all around the world. At any moment in time, they maybe updated or removed. Crawled documents are generally subject to a cleaning phase where some words are removed (very frequent words called stopwords) or transformed by removing all their affixes and taking into consideration only their stems. Also, there are duplicates, i.e. identical web pages, or quasi-duplicates to be removed. And this is not the whole story. Later the whole corpus is indexed, that means transformed in a format that allows efficient search. But this may also be transformed in a different format to represent a different kind of knowledge rather than simply the text contained in the documents. For example, a graph of pages linking one another can be extrapolated and used to calculate the PageRank [53] and eventually improve the search experience.
But, as we said before, web is not only about text search. This is why every kind of available data is crawled and used to improve the web experience and to support related commercial interests. Thanks to the knowledge extracted from query logs it is possible to suggest new queries to the user, thus helping him to disambiguate and/or refine queries. Browsing sessions can be mined to discover which are the end points of users’ navigations, i.e. the really interesting pages that can be recommended to drive the user quickly to his target. Also consider that most of the profits of commercial entities such as Amazon, come from lots of one-time sells, and not from few popular items bought by a large number of people. This is why it is important discover profiles that allow to give targeted recommendations or advertisements to different kinds of users. In a few words, lots of interesting but hidden information can be used to improve web browsing efficiency and efficacy.

The list of data mining applications applied to the web is endless. In any of them we can find many issues common to every sub-discipline or application of data mining. Many of these issues come from the peculiarities of web data itself. For instance, consider that the expression “web-scale” is used to address an extremely huge collection of data. Heterogeneity, distribution, freshness are other challenges arising in web search and in data mining as well, that we are going to describe in the next section.

1.3 Challenges in data mining

Most of the difficulties encountered by traditional data analysis techniques belong to one of the following challenges.

*Challenges due to user interaction.* User involvement into the knowledge discovery process poses may issues. The user needs a query language able to describe different data mining tasks and to enforce user’s constraints into the process. One of such constraints is the user background knowledge, on whose basis only interesting and novel patterns have to be extracted. These results require an effective presentation, that needs to be easily understandable despite the complexity of the mining tasks, and able to drive interactively the user along a fruitful path of the discovery process.

*Challenges due to the complexity and diversity of data types.* Traditional data analysis focuses on homogeneous data types. Unfortunately homogeneity is often an optimistic assumption for data happen to be complex and heterogeneous. A web page contains semi-structured text, images and links to other pages. XML data is fully structured with explicit parent-child relationships. DNA sequences have sequential and three dimensional information. GPS tracks contain temporal and spatial information. New techniques are need to mine such complex types of data taking into account diversity in information type (e.g. text versus hyper links), as well as diversity in their granularity (e.g. years versus weeks).
1.4. Contribution of this thesis

Challenges due to the origin of data and its size. Data happens to be very large, both in terms of physical size and in terms of dimensionality. Non trivial approaches, such as out-of-core algorithms, must be exploited in order to deal with such data. It is mandatory to take into account the limits and the opportunities given by modern computers in order make feasible the analysis of this huge data flow. Also, most of mining algorithms handle exponential search spaces, that become rapidly intractable with many kinds of data, such as gene expression data, where the number of features involved is measured in thousands. Worse than that, data may not be located in a single site, thus distributed algorithm are needed, raising new problems such as privacy. Finally, data is evolving over time. Not only it is preferable to mine data incrementally rather than restarting from scratch every time, but also the evolution patterns are of interest in many cases.

In this thesis, we deal with some of performance issues that relate to all of the above challenges. Performance is needed to allow prompt user interaction. Performance is needed to handle the large search space given by complex data types. Performance is needed to overcome the size of data.

We believe that high performance data mining is not only about writing fast code, but it is also about designing new algorithms, solving new problems, finding new interesting point of views. Also, high performance data mining often means making complex problems feasible.

1.4 Contribution of this thesis

This thesis is devoted to the design of high performance algorithms for the extraction of closed frequent itemsets from transactional databases.

In Section 2 we discuss the importance of a high performance point of view in the design of a pattern mining algorithm. We describe some of the main issues arising in such algorithms: computational complexity, data size and the opportunities provided by emerging computer architectures. For each of these issues, in this thesis we contribute a novel algorithm.

In chapter 3 we introduce the constraint-based mining paradigm. A powerful query language in form of constraints is provided to the analyst. A number of constraints allow him to define the portion of the search space of interest. We propose a new framework based on data-reduction where every possible conjunction of constraint can be exploited. Our goal is to show how high performance issues are fundamental in the design of any data mining algorithm. This is the first comprehensive algorithm of this kind. The core mining engine, is a module of our software called ConQuest. This allows the user to experience the KDD process from the data selection directly from relational databases, to the evaluation of the extracted association rules, via mouse-clicks. The general framework was published on the Data and Knowledge Engineering journal on February 2007 [10], while the ConQuest software was presented as a demo at the International Conference on Data
1. Introduction

Engineering on April 2006 [6].

Chapter 4 describes DCI-Closed: a new algorithm for mining closed frequent itemsets. This algorithm outperforms other state-of-the-art algorithms. It is the only algorithm that can mine dense datasets without maintaining the collection of closed itemsets mined so far, and, differently from FP-tree based implementations, it has high spatial locality in memory access patterns. It has other interesting features that allowed us to design both out-of-core and a parallel algorithms inspired by DCI-Closed. The algorithm DCI-Closed was published on the Transaction on Knowledge Discovery and Data Engineering on January 2006 [45].

When mining large datasets, most algorithms fail due to their excessive memory needs. In Chapter 5 we introduce the first algorithm for mining closed frequent itemsets in secondary memory. This problem was never solved before due to the difficulty of identifying closed itemsets in a partition of the original dataset. We show that it is easy to create sub-problem that can be solved in main memory, and that the problem of detecting non closed itemsets boils down to the trivial problem of sorting strings in secondary memory. This algorithm was published in the proceedings of the SIAM international conference on Data Mining on 2006 [46].

Finally, in Chapter 6 we discuss the opportunities and the challenges provided by modern computer architectures. Multiple cores are embedded in modern CPU, using non trivial cache hierarchies to access the main memory and equipped with a large set of SIMD instructions. This translates to a significant amount of processing power that needs the contribute from algorithm designers. Parallelism becomes mandatory and it must be coded explicitly. Multiple cores competing for the same shared cache need a provident design of memory access patterns. Also, SIMD instructions can help in improving performance. Inspired by this setting, we proposed a multi-threaded algorithm for mining closed frequent itemsets called MT-Closed. We show that it is possible to partition the mining problem into completely independent tasks that can be assigned to a pool of threads. We analyze different task assignment and data partitioning strategies, including a non trivial one implementing a work-stealing strategy. While this strategy is not novel, its application in pattern mining algorithm is rare. MT-Closed is accepted for publication at the IEEE International Conference on Data Mining on 2007 [47].

Enjoy reading!
The importance of high performance in frequent itemsets mining

In this chapter we contextualize the contributions of this thesis. We present the problem of mining frequent itemsets from databases. We discuss some of its main issues, such as the computational complexity, the data size and the opportunities given by modern computer architectures. This thesis contributes to all the three issues. While discussing these, we also introduce some notation, basic concepts and fundamental algorithms.

We also want to show that high performance is a flavor that any well-designed algorithm should have. In fact, high performance algorithms success where other naïve algorithms fail.

2.1 Introduction to frequent itemsets mining

Association analysis is a central task of data mining. It aims at finding strongly correlated sets of events from databases. Association rules are able to fit many application fields, such as market basket data analysis, web log analysis, credit card analysis, call record analysis etc.

An association rule is expressed as an implication between two sets of events:

\[ r : A \rightarrow B. \]

In the market basket analysis setting, our dataset is the set of transactions issued by customers, and every transaction consists of the set of items purchased. An association rule will regard two sets of items that where bought together, e.g.:

\[ r_1 : \{\text{diapers}\} \rightarrow \{\text{milk, beer}\}. \]

There are two basic measures of a rule’s interestingness, support \(\sigma\) and confidence \(\gamma\). The support of a rule \(\sigma(r)\) is defined as the ratio of transactions in the dataset that contain all the elements \(\{A \cup B\}\) in the rule. The confidence is the ratio of transactions that contain also \(B\) among the ones containing \(A\). These can also be expressed in terms of probabilities: \(\sigma(r) = P(\{A \cup B\})\) and \(\gamma(r) = P(B|A)\).
An association rule with high support regards many transactions of the database, and therefore, the support tells us about the frequency of the rule. On the other hand, the confidence tells us about the strength of the implication. If the rule \( r \) has low confidence, it means that only a small number of times \( B \) occurs together with \( A \), while there are many transactions that contain \( A \) but not \( B \).

The user is responsible for choosing a good pair of thresholds \( \sigma \) and \( \gamma \), according to which a set of frequent and confident rules, that is having support at least \( \sigma \) and confidence at least \( \gamma \), has to be extracted.

The mining of association rules is run in two steps: (a) first all frequent itemsets are extracted and then (b) these are used to generate confident rules. Given a frequent itemset \( X \), i.e. that occurs at least \( \sigma \) times in the database, any rule in the form \( r : A \rightarrow B \) with \( X = \{A \cup B\} \) is frequent as well. For each discovered frequent itemset \( X \), we need to generate all the frequent rules \( r : A \rightarrow B \) with \( X = \{A \cup B\} \), and then to check whether these are confident or not.

While the second step of the association analysis is trivial, the first one is very complex and time consuming. Moreover frequent itemsets are by themselves very important. In fact, many different kinds of frequent patterns have been studied, such as frequent sequential patterns [61] in customers’ purchases, frequent episodes [51] in network logs, frequent trees [73] and frequent sub-graphs [37] in chemical compounds datasets, and many others.

Since mining frequent itemsets is an important and challenging task, in the remainder, we will discard the generation of association rules and we will focus on the frequent pattern extraction only.

### 2.2 Frequent itemsets mining algorithms

We formalize the problem of mining frequent itemsets as follows:

**Definition 2.1 (FIM: Frequent Itemset Mining Problem)**

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

The main difficulty of FIM algorithms comes from the large size of the search space. In principle, every itemset in powerset \( \mathcal{P}(\mathcal{I}) \) is potentially frequent, and a whole scan of the dataset is required to calculate its exact support. As usual we will use a lattice of itemsets, where node are in lexicographical order, to visualize such search space as in Figure 2.1.
2.2. Frequent itemsets mining algorithms

the dataset \( \mathcal{D} \)

<table>
<thead>
<tr>
<th>TID</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{b, d}</td>
</tr>
<tr>
<td>2</td>
<td>{a, b, c, d}</td>
</tr>
<tr>
<td>3</td>
<td>{a, c, d}</td>
</tr>
<tr>
<td>4</td>
<td>{c}</td>
</tr>
</tbody>
</table>

More formally, the problem of finding the number of frequent itemsets is \#P-hard [29]. The class of \#P problems is the counting counterpart of usual decision problems. For instance, a \#P problem is \#SAT: “How many different variables assignments will satisfy a given DNF formula?”. The classes of \#P, \#P-hard and \#P-complete problems were first introduced in [65], and consider that many pattern mining tasks fall in to the toughest class of \#P-complete problems [72].

Apriori [1]

The first proposed FIM algorithm is APRIORI. It exploits a bottom-up level-wise exploration of the lattice of frequent itemsets. During a first scan of the dataset, all the frequent singletons, denoted with \( \mathcal{L}_1 \), are found. Next, \( \mathcal{L}_1 \) is used to find the set of frequent 2-itemsets \( \mathcal{L}_2 \), and so on until all the frequent patterns have been discovered.

Each iteration is composed of two steps: candidate generation and support counting. During the first step, a collection of possibly frequent itemsets \( \mathcal{C}_k \) of length \( k \) is generated, and their actual support is calculated with a single scan over the dataset during the second step. In principle, it would be easy to generate at each iteration all the possible \( k \)-itemsets given that \( \mathcal{I} \) is known. In this way, we would have an extremely large collection of itemsets that have to pass the support counting phase. The APRIORI algorithm uses a simple property in order to reduce the number of
Table 2.1: Symbols and notation used in this thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c, i, j, . . .</td>
<td>items are denoted with lower-case letters</td>
</tr>
<tr>
<td>I, X, Y, Z, . . .</td>
<td>itemset are denoted with upper-case letters</td>
</tr>
<tr>
<td>X = {abc . . .}</td>
<td>an itemset (we drop commas)</td>
</tr>
<tr>
<td>X = &lt;abc . . .&gt;</td>
<td>sorted itemset (according to some (&lt;))</td>
</tr>
<tr>
<td>D</td>
<td>transactional input dataset</td>
</tr>
<tr>
<td>D_k</td>
<td>horizontal dataset at iteration k</td>
</tr>
<tr>
<td>D_X</td>
<td>dataset projected over the itemset X</td>
</tr>
<tr>
<td>BM</td>
<td>vertical bitmap representing the dataset</td>
</tr>
<tr>
<td>(\sigma(\cdot))</td>
<td>support function and minimum support threshold</td>
</tr>
<tr>
<td>(\gamma(\cdot))</td>
<td>confidence function and minimum confidence threshold</td>
</tr>
<tr>
<td>(L, C, R)</td>
<td>the collection of frequent, closed and valid itemsets</td>
</tr>
<tr>
<td>(C_k, L_k, R_k)</td>
<td>candidate, frequent and valid itemsets of length k</td>
</tr>
<tr>
<td>c(\cdot)</td>
<td>closure operator</td>
</tr>
<tr>
<td>g(X)</td>
<td>set of transactions supporting the itemset X</td>
</tr>
<tr>
<td>f(T)</td>
<td>intersection of the transactions in the set T</td>
</tr>
<tr>
<td>(\hat{M})</td>
<td>maximum memory available</td>
</tr>
</tbody>
</table>

candidates at each iteration:

**Apriori Property** (or **downward closure**): all the subsets of a frequent itemset are frequent as well.

The algorithm exploits a pruning of the search space using the reverse implication of such property. If \(X \in C_k\) has a subset of length \(k - 1\) that does not belong to \(L_{k-1}\), i.e. this subset is not frequent, then \(X\) cannot be frequent and it can be removed from the candidate set \(C_k\). Note that when removing an itemset \(X\) we can also discard all its supersets. By reducing the number of candidates, the algorithm strongly reduces the cost of the support counting phase. This provides a stopping criterion: if \(L_k\) is empty at some iteration \(k\), then all the frequent itemsets have already been discovered. The pseudo-code of **APRIORI** is reported in Algorithm 1.

Also consider that the same property can be used to remove useless information from the dataset, for instance every item that does not belong to \(L_1\) can be pruned since it will never participate to the support of a frequent itemset.

The name of the algorithm comes from the fact that it uses the **prior** knowledge about itemsets already found \((L_{k-1})\), in order to discover new frequent itemsets \((L_k)\).

In the following sections we discuss three main issues arising in frequent pattern mining: the computational complexity, the size of the data and the opportunities coming from modern computer architectures. We also introduce some fundamental
2.3. Issue #1: computational cost

The cost of a FIM algorithm comes from two major contributions: the number of candidate itemsets evaluated, and the size of the dataset.

The number of candidate itemsets, i.e. the size of the search space, depends on the number of frequent singletons $|\mathcal{L}_1|$, and it can be considered proportional to $2^{|\mathcal{L}_1|}$, which is the cardinality of the powerset of $\mathcal{L}_1$. This means that this size will increase exponentially as the $\sigma$ decreases. And, the larger the set of candidates, the larger the cost of the support counting phase.

The number of transactions present in the dataset affects the support counting. Each transaction has to be checked in order to understand whether it supports a candidate or not. Note that this cost also increases non trivially with the transaction length. Finally, in Apriori, the dataset is supposed to be stored on disk, and thus each full scan executed is considerably expensive.
Reducing the number of candidate itemsets is very difficult, basically because it is hard to predict the support of an itemset. Therefore, efficient data structures are needed to manage large sets of candidates.

On the other hand, a very elegant approach allows to reduce the number of transactions to be processed at each iteration (and their length). This is done by changing the lattice traversing strategy, moving from level-wise to depth-first and embracing a *divide et impera* philosophy.

**Eclat** [76] and **FP-Growth** [32]

Both Eclat and FP-Growth share the same idea of subdividing the search space on a prefix base. After the first scan, the original problem can be subdivided into \(|L_1|\) independent sub-problems, where \(L_1 = \{a, b, \ldots, z\}\) is the set of frequent singletons sorted according to some order \(<\). *Having introduced the order \(<\), hereinafter, we assume any itemsets \(X\) to be a sorted set \(X = \langle x_1, \ldots, x_{|X|}\rangle\).*

The mining proceeds as follows: first find the itemsets starting with the frequent item \(a\), then the ones starting with the second frequent item \(b\), and so on. Of course each sub-problem can be recursively sub-divided on the basis of increasing length prefixes: first mine the itemsets starting with \(\{a, b\}\), then \(\{a, c\}\), etc. This corresponds to a depth-first visit of the lattice.

In this way, the algorithm does not have to handle a large set of candidates \(L_k\), it must store only the information regarding nodes on the path from the root to the current itemset. This avoids any concern related to the maintenance of the collection of candidates.

In order to reduce the cost of support counting, the algorithm stores at each node of the visit a *projected*, i.e. shrunken dataset stored in main memory that contains only the useful transactions. For instance, if we are mining the itemsets starting with the item \(a\), then we can disregard all those transactions that do not contain \(a\). Later, when mining the itemsets starting with \(b\), then we can also discard every occurrence of the item \(a\). This procedure can be repeated recursively and this idea can be generalized to itemsets, rather than single items. Given an \(k\)-itemset \(X = \langle x_1, \ldots, x_k\rangle\), we denote with \(D_X\) a projection of the dataset \(D\) defined as follows:

\[
D_X = \{t' = t \setminus \{i \in t \mid i \preceq x_k\} \mid t \in D \land X \subseteq t\}
\]

Note that we removed also any item of \(X\) in \(D_X\) since we know that \(D_X\) is built from transactions that contain \(X\), i.e. storing \(X\) is redundant.

These projected datasets are stored by the two algorithms using different data structures.

Eclat adopts a *vertical format*. For each item, it stores the list of transaction identifiers containing that item. This kind of inverted list is called *tid-list*. To create a projection of the dataset on an item \(i\), it is sufficient to intersect every tid-list with the one of \(i\), and, at each step of the visit a new item is used to build a new
2.3. Issue #1: computational cost

projection. Note that with this vertical representation, the support of an itemset is obtained via tid-lists intersections. In fact, the support of an itemset is calculated as the cardinality of the intersection of the items tid-lists that compose the itemset.

Later, the same authors proposed a new representation for these tid-lists, named diff-sets [74], that improves their space occupancy. This allows faster intersections and a faster algorithm called DECLAT.

FP-GROWTH differs for its orthogonal data representation. The dataset is stored in horizontal format, that is transaction by transaction, but in a new data structure, called FP-tree, which is very similar to a trie. Since frequent prefixes are shared among the transactions stored along the paths of the trie, this turns out to be quite compact. Projections are created thanks to an auxiliary data structure that links all the occurrences of the same item in the FP-tree. Using this, it is easy to traverse each transaction in the tree containing a given item and, at the same time, to build a new projected dataset.

This data structure proved to be a very flexible, and it was used to mine almost every other kind of pattern, like sequences [58] and graphs [69].

![Figure 2.2: A path of depth first visit of frequent itemsets lattice. Each node along the path is accompanied by the corresponding projection of the dataset.](image)

The great advantage of this divide et impera approach is given by the dramatical reduction of the amount of information to be processed at each operation. Each recursive projection removes a large portion of the original dataset. Thus, a dataset smaller than the original accompanies each node of the lattice as shown in Figure 2.2. Its reduced size speeds up the support counting step. Not to mention that a more compact data structure provides improved spatial and temporal locality of the accesses to the dataset that it is now resident in main memory. In Algorithm 2 we give the pseudo-code of these two typical depth first algorithms.

In Chapter 4 we propose DCI-CLOSED [45], a new algorithm that overcomes the
Algorithm 2 ECLAT and FP-GROWTH: two depth first algorithms.

1: function MINE(D, σ)
2: if |D| < σ then ▷ the empty set is a special case
3: return ∅ ▷ the collection of frequent itemsets
4: end if
5: $L_0 \leftarrow \{\emptyset\}$ ▷ store the dataset in main memory
6: if this is ECLAT then
7: $D \leftarrow$ vertical-format(D)
8: else this is FP-GROWTH
9: $D \leftarrow$ horizontal-format(D)
10: end if
11: MINE-NODE($\emptyset$, $D$, $\sigma$, $L$) ▷ visit the lattice starting from $\emptyset$
12: return $L$
13: end function

14: procedure MINE-NODE (X, $D_X$, $\sigma$, $L$)
15: $I_X \leftarrow$ get-frequent-singletons($D_X$, $\sigma$)
16: for $i \in I_X$ do
17: $Y \leftarrow X \cup i$ ▷ generate a new frequent itemset
18: $L \leftarrow L \cup Y$
19: $D_Y = \text{project}(D_X, i)$ ▷ build a new projection
20: MINE-NODE($Y$, $D_Y$, $\sigma$, $L$) ▷ recursive mining
21: end for
22: end procedure

computational complexity issue thanks to a bitmap representation of the dataset, and to a different duplicate detection technique.

2.4 Issue #2: data size

The second issue regards the feasibility of handling the amount of information to be processed. When the algorithms we discussed before are to be materialized on a real machine, then they have to handle its physical constraints.

For instance, the amount of available memory is a significant limit. If an algorithm is not conscious of this limit, it will be likely to break it. In case this happens, the performance will dramatically degrade because of continuous memory swaps carried out by the operating system.

These kinds of problems arise very often with data mining algorithms. Not only because of the size of the datasets, but also because of the large data structures used.
2.4. Issue #2: data size

The first Apriori implementation assumed the dataset to be resident on disk. Thus, each iteration comprises a full scan of the datasets, that harms the efficiency of the mining. And, it is often the case where the dataset cannot be loaded in main memory because of its large size.

A number of algorithms have been designed aiming to overcome this problem. The idea is to split the initial dataset into smaller sets of transactions, such that each of them can fit in the available memory. The Partition algorithm is based on the following property: a frequent itemset must be frequent in at least one partition of the database. It mines sequentially each partition one at a time, thus being guaranteed to find which are the frequent itemsets after one complete pass over the database. A second scan is required to collect the supports of those itemsets that were not frequent in every partition, since the support of infrequent itemsets is not calculated.

A more aggressive algorithm is Sampling: it bets on the chance to discover all the frequent itemsets in a single pass over the dataset. It first extracts all the frequent itemsets from a small sample of the database that can be handled entirely in main memory. Authors also show how to use Chernoff bounds to estimate what is the least number of transactions needed. Then the transactions outside the sample are used to compute the actual support of the itemsets found in the sample. The algorithm also stores some additional information that allows to understand whether all the frequent itemsets have been discovered or any miss occurred. In the latter case, a second scan the dataset is used to extract all the remaining frequent itemsets. The authors show that in many cases it is possible to compute all the frequent and confident association rules in one step, and when a second pass over the dataset is needed, this is not significantly expensive.

Even depth-first algorithms fail in dealing with large databases. In the previous section we have seen how they reduce the amount of data processed at each step thanks to recursive projections. However, they have to store all those projected datasets along the path from the root to the current node of the visit. Their cumulative size is very likely to be too large to be managed in main memory. In our experiments we found that state-of-the-art algorithms, such as LCM and FP-Close, fail to mine the USCensus1990 (500MB) and Synth2GB (2GB) datasets with a quite large minimum support threshold of 40%, on a machine with 2.5GB of memory. Also in the case of depth-first algorithms, a number of out-of-core alternatives have been developed such as DiskMine. The algorithm is based on partitioning of the search space able to reduce the size of projected datasets. We describe this algorithm more in detail in Section 5.2.2.

In Chapter 5 we introduce a novel algorithm, called OOC-Closed, for a related pattern mining problem. This is actually the first algorithm solving such problem. A detailed analysis shows that an inevitable merging of local results can

\footnote{Details on the datasets used in this thesis are given in Appendix A.}
be as easy as sorting strings.

2.5 Issue #3: parallel mining and emerging computer architectures

The FIM problem is very demanding in terms of computing power. Mining large datasets or using low minimum support thresholds increases exponentially the running time of every algorithm.

Consider a web search engine as we did in Section 1.2: it would be unacceptable to wait for hours before having an answer to a query. The effectiveness of popular web search engines is due to the large number of machines working in parallel to answer the incoming stream of queries.

The KDD process depicts a mining engine able to provide almost instantaneous answers and continuous interaction. In the very same way, parallel computing is considered the basin from which to borrow the large computing power needed to materialize such picture. In fact, there are interesting infrastructures, such as BOINC [2] (Berkeley Open Infrastructure for Network Computing), that suite for massively parallel tasks. BOINC is fruitfully used by important projects such as Seti@Home [3], Climate@Home, Rosetta@Home, Folding@Home, Einstein@Home and many others [2].

A parallel computing approach becomes more and more interesting, and in a way almost tangible, if we consider that a modern CPU is itself a parallel machine. Dual core processors have two independent CPU that can run in parallel with little or no interferences. Also consider that quad-cores are already available and that soon CPU with tenths of cores will be in every home PC.

Due to the gap between CPU and memory speed, new and more complicated cache hierarchies will accompany multi-core processors. At least three levels of cache memory seem mandatory, and some internal levels may be shared to only a subset of all the cores.

Finally, consider that the instruction sets of modern CPUs is growing with a large number of SIMD instructions, thus providing additional chances for parallelism.

A particular trend is given by game devices, i.e. video cards and consoles. On one hand, GPU have become even more complex than GPUs. NVIDIA proposes the Tesla processor, which has 128 (shader) processor and 1.5GB of dedicated memory in the entry level configuration. The IBM Cell processor, which is embedded in Playstation3, has a very interesting SMP architecture made with one PowerPC core and eight specialized processing units, called synergistic processing elements, having 128 bit registers and private storage area with no cache memory. It is not by chance that such amount of power is being used in many ways: GPUs for sorting billions of

\[\text{For the projects cited here see } \text{http://www.climateprediction.net/}, \text{http://boinc.bakerlab.org/rosetta/}, \text{http://folding.stanford.edu/} \text{ and http://einstein.phys.uwm.edu/}.\]
database records [24], for mining data streams [25], for executing complex database operations [18] and clusters of Playstation consoles for academic purposes [3].

We can thus recognize three main trends: increase of the number of core, introduction of SIMD instructions and increasing complexity of cache hierarchies. Behind such trends and such high performance SMP machines, new challenges are hiding. New computational layouts will be required for next generation algorithms. New data mining algorithm will have to be parallel. Being data intensive, they will have to use different memory access patterns, overtaking cache conscious/oblivious algorithms toward new paradigms of cooperative caching [15]. Also, they will use advanced SIMD instructions, which in turn require a proper design of these algorithms.

Emerging architectures pose new challenges a new opportunities for data mining algorithm designers, which will have to look at algorithms from new perspectives. These new algorithms will provide the capability to attack new problems and to tackle new types of data.

Chapter 6 introduces a novel algorithm called MT-CLOSED [47], which is the first one trying to take into account all these three emerging trends in modern computers architectures.

2.6 Other issues

What we have describe so far is not a comprehensive list of the may aspects that have to be taken into account when designing a good mining algorithm.

Another important issues is the size of the output. When using low minimum support threshold, the number of extracted patterns rapidly becomes far from tractable. To face this problem two approaches were adopted. The first is to allow the user to identify a region of interest in the search space using a set of constraints. This is discussed in Chapter 3. Another complementary approach is to mine a condensed representation of the output. Closed itemsets are one of such compact representations: they are order of magnitudes less than frequent itemsets but they allow to reconstruct the whole collection of frequent itemsets. Closed itemsets are the most important topic in this thesis. Chapters from 4 to 6 will deal with different aspects of the closed itemsets mining problem.

As a representative of the other anonymous issues, we discuss here the problem of the data dimensionality. We have already seen how the number of frequent itemsets affects the size of the search space. This is not only a reason explaining the cost of a FIM algorithm. A large search space may make the mining task unfeasible, and causes all the algorithms we described above to fail.

The number of candidates increases exponentially as the minimum support threshold. In many papers, this issue is often referred as the scalability w.r.t. $\bar{\sigma}$. However,
in many settings the size of the search space happens to be huge even with reasonable minimum supports. This is the case of certain kinds of datasets, for instance DNA micro-arrays, where we are looking for correlations among genes activities in dataset having a small number of transactions, i.e. experiments, and a very large number of items, i.e. genes’ responses. This makes the number of candidates explode.

The size of search space is not any more a costly aspect of the problem but a feasibility issue. In fact, it is not rare to see \textit{Apriori} failing because it is not able to handle that many candidates in memory. On the other hand, depth first algorithm will run for ages because of the huge number of projections to be created during the computation.

The solutions to such problems are not found with a restyling of the algorithm or a smart engineering of the code, but exploring completely new algorithms or trying to find equivalent problems easier to be solved. In this sense, high performance data mining is not only about designing fast algorithms, but also about finding new algorithms and new solutions to new problems.

\textbf{Carpenter} \cite{54}

The authors of \textit{Carpenter} had the intuition to reshape the original dataset into a new and more appealing one. They transposed the dataset swapping the role of transactions identifiers with the one of the items. The result is a dataset with a small number of literals, having small set of candidates that can thus easily mined. The extracted collection of $\textit{tid-sets}$ can easily be re-transposed in order to come back to our usual collection of frequent itemsets. This is a case, were in order to make the original problem tractable, it had be reformulated in a “better” way.

\textbf{Other algorithms}

There many other algorithms able to solve the FIM problem, taking into account other additional factors that can affect a successful mining. For instance, some of them can recognize whether a dataset is dense, i.e. with many correlated transactions, or sparse and adopt different strategies or data structures. And many other variants are still under development.

The only conclusion we can draw is that there is not a winning algorithm. Data may occur in different shapes, and this shape determines what is the best algorithm. Even if \textit{Apriori} is the oldest and generally the slowest, there are cases, i.e. datasets, where it can beat state-of-the art algorithms.

As long as different kinds of data will emerge, there will be room for new outperforming algorithms.

\footnote{\textit{Carpenter} actually mines closed itemsets, but this does not makes any difference for the scope of our discussion.}
2.7 Conclusions

In this chapter we introduced some preliminary concepts and algorithms for the extraction of frequent itemsets from transactional databases.

In particular we stressed three important aspects of this kind of algorithms: computational complexity, data size and the opportunities given modern computer architectures. Designing a robust algorithm means to deal with these issues. This means that designing an algorithm from an high performance point of view is mandatory for the developing of a good algorithm in general.

We also tried to show that high performance data mining is not only about code engineering, but is also a vantage point that allows to find new algorithms and new solutions to new problems. In Chapter 3 we show how these issues arise in the constrained pattern mining paradigm.

Moreover, we introduced the main contribution of this thesis. Each of the chapters from 4 to 6 deals with one of the above three issues, but dealing with the related problem of mining closed frequent itemsets.
2. The importance of high performance in frequent itemsets mining
3

ConQuEst: a Constraint-based Querying System for exploratory pattern discovery

3.1 The constrained pattern mining paradigm

We have seen that a wide spectrum of algorithms have been designed to solve the FIM problem. At this point we would like to come back to the idea of the knowledge discovery as an iterative and interactive process with the user. We will use our experience in constrained pattern mining to show how some of the aforementioned issues arise even in this particular data mining setting and how they can be solved effectively. Again, high performance data mining leaded us to discover new techniques and algorithms able to accomplish tasks otherwise intractable.

Once, the user enters the stage we must take into consideration a few important aspects. First, it is often necessary to use low support thresholds to discover interesting and unexpected patterns. Generally, the most frequent patterns are well known to a domain expert that already has a good background knowledge. Of course, the lower the minimum support, the larger the number of extracted patterns. It is not unusual that this collection of frequent patterns is too large to be handled directly by the analyst. This is sometimes referred as a second order data mining problem: it seems necessary to mine among the patterns already mined.

Also, frequency is not always a good selection criterion. It is often the case that one would like to be able to focus the mining process on a particular portion of the search space. Or, said in other words, it is important to have some additional expressiveness in order to define what is interesting and what is not.

In order to solve these problems, the framework of constrained pattern mining was developed. The idea is introduce a set of constraints that an itemset may satisfy or violate. The analyst will use conjunction of constraints to specify the properties of the patterns of interest. Indeed, these constraints can be used fruitfully by the algorithm to restrict the search space and to speed up the computation. Formally, we define the constrained pattern mining problem as follows.
Definition 3.1 (CFIM: Constrained Frequent Itemset Mining)
A constraint is a function \( C : 2^I \rightarrow \{\text{true}, \text{false}\} \). We say that an itemset \( X \) satisfies a constraint, i.e. it is valid, if and only if \( C(I) = \text{true} \). Given a transactional dataset \( D \), a minimum support threshold \( \sigma \) and a set of constraints \( C = \{C_1, \ldots, C_n\} \), the Constrained Frequent Itemset Mining Problem requires to discover all the itemsets \( X \) such that \( \sigma_D(X) \geq \sigma \) (they are frequent) and \( C_1(X) \land \ldots \land C_n(X) = \text{true} \) (they are valid).

Several classes of constraint and their properties have been identified: anti-monotone [1] (the minimum support is an anti-monotone constraint), succinct [40], monotone [12], convertible [56], loose anti-monotone [9]. In Figure 3.1 we give a picture of all the classes of constraints studied so far. For each of those classes there exists a specialized algorithm which is able to take advantage of its peculiar properties.

Example 3.1 (An example of constraint) Let’s come back to the market basket analysis setting. We can associate to each item a fixed price attribute. Then, the constraint “\( C = \text{sum}(X, \text{price}) > 100\$\)” signifies that we are interested only in those co-occurring set of items such that their average price is greater than 100\$. This constraint is monotone.

Recall that we always assume that a minimum support is always required. It is clear that the most frequent itemsets are in the lower part of the lattice (shorter itemsets), while the itemsets satisfying a constraint in Example 3.1 are in the upper part (longer itemsets). Thus the solution of the problem is exactly in the middle, meaning that an algorithm must traverse (either from the top or from the bottom) a large part of invalid itemsets before it can reach the region of interest in the search space.
This is even worse with other kinds of constraints such as loose anti-monotone ones. These are related with statistical functions, such as the variance, and they give a jagged contour to the region of valid itemsets: it is impossible to avoid the traversal of many invalid itemsets in order to reach the valid ones.

In this context high performance becomes very important. First, high performance is required to allow a prompt user interaction with the user. Secondly, it is mandatory to push the properties of each constraint into the algorithm design restrict the search space, which would otherwise be too large to be handled by traditional or naïve algorithms.

Some of the issues discussed so far arise here with greater strength. The use of lower minimum support threshold creates a larger number of candidates meaning an increased number of support calculations. Therefore, we need to reduce the computational cost of the problem. Also, since typical datasets are large, and the collection of candidates is larger then usual, the memory is still an issue. A good algorithm must be able to use a limited amount of memory, trying to use constraints to focus only on the interesting portion of the data. Finally, the minimum support threshold is linked to the dimensionality of the problem. In this case the problem is worse: not only a large number of candidates has to be visited, but most of them is invalid and therefore useless.

Because of these issues, a naïve post-processing algorithm that first computes all the frequent itemsets and then filter only the valid ones would be unfeasible. The size of the problem is intractable.

In this chapter we introduce an innovative framework for exploiting any conjunction of constraints. Instead of a new specialized algorithm for a given class of constraints, we provide the possibility to use at the same time any combination of constraints even if they belong to different classes. Our approach is based on a data-reduction technique. The resulting algorithm addresses all the high performance issues we discussed in the previous chapter (except parallel mining). We also present a comprehensive software tool for the mining of association rules from databases. This software is called ConQueSt.

3.2 A review of constraint-based algorithms

Constraint-based frequent pattern mining has been studied as a query optimization problem, i.e. developing efficient, sound and complete evaluation strategies for constraint-based mining queries. To this aim, properties of constraints have been studied comprehensively: anti-monotonicity \cite{1}, succinctness \cite{10}, monotonicity \cite{12}, convertibility \cite{56}, loose anti-monotonicity \cite{9}, and on the basis of such properties efficient computational strategies have been defined. However, the proposed strategies cannot be exploited altogether in a comprehensive algorithm. A preliminary effort to propose a general framework for constrained frequent itemsets mining is \cite{10}: ConQueSt is the actual realization of that framework, and thus it
is the first software that is able to deal with all of these classes of constraints at the same time.

In the following, together with an exhaustive presentation of the different constraints classes, we review the algorithms proposed to exploit constraints properties in order to reduce the computational cost of data mining algorithms by means of search-space and data reduction.

The first interesting property is the anti-monotonicity, which was already introduced with the APRIORI algorithm, since the minimum frequency is actually an anti-monotone constrain.

**Definition 3.2 (Anti-monotone constraint)** Given an itemset $X$, a constraint $C_{AM}$ is anti-monotone if $\forall Y \subseteq X : C_{AM}(X) \Rightarrow C_{AM}(Y)$.

This the anti-monotonicity property can be used to reduce both the search space and the data. A large portion of the search space can be pruned whenever we meet an infrequent itemset $X$ then no superset of $X$ can satisfy minimum frequency constraint. Additionally, many well known related properties can be used to shorten the transactions in the dataset. These properties boil down to the following: if an item in a transaction is not included in $k$ frequent itemsets of length $k$ supported by that transaction, then it will not be included in any frequent itemset of length greater than $k$. These items can be discarded after the $k$-iteration of a level-wise algorithm. Eventually the whole transaction can be pruned from the dataset.

**Definition 3.3 (Succinct constraint)** An itemset $I_s \subseteq I$ is a succinct set, if it can be expressed as $\sigma_p(I)$ for some selection predicate $p$, where $\sigma$ is the selection operator. $SP \subseteq 2^I$ is a succinct power-set, if there is a fixed number of succinct sets $I_1, I_2, ..., I_k \subseteq I$, such that $SP$ can be expressed in terms of the strict power-sets of $I_1, I_2, ..., I_k$ using union and minus. Finally, a constraint $C_s$ is succinct provided that the set of itemsets satisfying the constraint is a succinct power-set.

Informally, a succinct constraint $C_s$ is such that, whether an itemset $X$ satisfies it or not, can be determined based on the singleton items which are in $X$. An example of succinct constraints, is $C_s(X) \equiv "X$ contains items of type food”

This class of constraints was introduced with the CAP [10] algorithm. In general it is possible to understand if no supersets of a given itemset can satisfy the constraint, and remove those supersets from the search space. However, since supersets of invalid itemsets cannot be pruned, this strategy does not provide an effective reduction as anti-monotone ones. On the other hand, constraints that are both anti-monotone and succinct can be pushed once and for all at pre-processing time, by removing invalid items from the dataset.

**Definition 3.4 (Monotone constraint)** Given an itemset $X$, a constraint $C_M$ is monotone if: $\forall Y \supseteq X : C_M(X) \Rightarrow C_M(Y)$. 
3.2. A review of constraint-based algorithms

Handling monotone constraints in conjunction with the minimum frequency constraint (that it is always present in any query) and with other anti-monotone constraints, turns out to be very difficult due to their symmetrical behaviour: valid patterns w.r.t. the frequency constraint are in the lower part of the itemsets lattice, while itemsets that satisfy \( C_M \) are in the upper part. A typical bottom-up as well as a top-down visit introduced by the Dual-Miner \cite{12} algorithm will inevitably traverse many invalid itemsets w.r.t. one of the two classes of constraints.

Also consider that only little work can be saved due to the \( C_{AM} - C_M \) trade-off: if an itemset does not satisfy the monotone constraint, we could avoid the expensive frequency count for this itemset. On the other hand, if the same itemset was actually infrequent, it could have been pruned with its supersets from the search space, thus saving many more frequency counts.

Among the many algorithms \cite{59, 12} introduced for mining frequent patterns under a conjunction of monotone and anti-monotone constraints, a completely orthogonal approach was introduced in \cite{8, 7}. In fact, a transaction that does not satisfy the monotone constraint \( C_M \), can be removed since none of its subsets will satisfy \( C_M \) either, and therefore the transaction cannot support any valid itemsets. This data reduction, in turns reduces the support of other frequent but invalid itemsets thus reducing the search space and improving anti-monotone pruning techniques. This virtuous circle, and in general the synergy between data reduction and search space reduction, will inspire our algorithmic framework.

**Definition 3.5 (Convertible constraints)** A constraint \( C_{CAM} (C_{CM}) \) is convertible anti-monotone (monotone) provided there is an order \( R \) on items such that whenever an itemset \( X \) satisfies \( C_{CAM} \) (violates \( C_{CM} \)), so does any prefix of \( X \). If a constraint is both convertible monotone and anti-monotone, then it is called strongly convertible.

**Example 3.2 (\( \text{avg}(X.price) > 100\$ \) is a convertible constraint)** Let \( R \) be the descending price order, then \( \text{avg}(X.price) > 100\$ \) is a convertible anti-monotone constraint since \( C_{CAM}(X) \Rightarrow C_{CAM}(Y) \) where \( Y \) is prefix of \( X \). In fact, the constraint is convertible monotone with the inverse order of \( R \), and thus it is strongly convertible.

The class of convertible constraints was introduced in \cite{56}, where authors propose two FP-GROWTH based algorithms, named Fic-A and Fic-M. During the construction of the initial FP-tree, items are ordered according to \( R \), and then a slightly modified FP-GROWTH starts the mining. In presence of a \( C_{CAM} \) constraint, Fic-A stops the visit of the search space whenever an invalid itemset is found. While, in presence of a \( C_{CM} \) constraint, no pruning is performed. The Fic-M can only save monotone checks for the supersets of valid itemsets. Unfortunately, the order \( R \) may significantly affect the performances of the algorithm. Using an ordering of items different from the ascending frequency one, may slow down the algorithm by a factor of 10.
The same classes of constraints can also be exploited in a level-wise framework, as we have shown in [10]: if transactions are sorted according to the same order \( \mathcal{R} \), some advanced pruning strategies may take place, since \( C_{\text{CAM}} \) and \( C_{\text{CM}} \) impose stronger requirements for an item of a given transaction to be useful in the subsequent iterations of a level-wise strategy.

**Definition 3.6 (Loose Anti-monotone constraint)** Given an itemset \( X \) with \( |X| > 2 \), a constraint is loose anti-monotone (denoted \( C_{\text{LAM}} \)) if: \( C_{\text{LAM}}(X) \Rightarrow \exists i \in X : C_{\text{LAM}}(X \{i\}) \)

We introduced the class of loose anti-monotone constraints in [9]. These are a proper superset of convertible constraints, also related to many interesting statistical functions, such as variance, standard deviation, mean deviation, etc. which cannot form convertible constraints. Every constraint in this class can be exploited in a level-wise computation by means of data reduction. In fact, if a at level \( k > 1 \) a transaction is not a superset of any valid itemset of size \( k \), then it will not contain any valid larger itemset.

**Example 3.3 (variance(\(X\).price) < 100$ is a loose anti-monotone constraint)** Given an itemset \( X \), if it satisfies the constraint so trivially does \( X \{i\} \), where \( i \) is the element of \( X \) which has associated a value of price which is the most far away from \( \text{average}(X\text{.price}) \). In fact, we have that \( \text{variance}((X \{i\}).\text{price}) \leq \text{variance}(X\text{.price}) \), until \(|X| > 2\).

In Table 3.1 we give a classification of many commonly used constraints.

### 3.3 A new data-reduction framework for constrained pattern mining

In Table 3.2 we summarize the aforementioned classes of constraints and some representative algorithms that exploit their properties. For each constraint, there exist both data reduction and search space reduction algorithms. Recall the our system should be able to answer conjunctive queries possibly containing many constraints belonging to different classes.

Unfortunately, most of the search space reduction based strategy cannot be exploited at the same time. For instance, **Dual-Miner** top-down search can hardly be adapted to traditional bottom-up strategies, and reordering items as **Fic-A** and **Fic-M** algorithms may not be possible in presence of multiple convertible constraints requiring different orderings.

On the other hand, all the other data reduction strategies are orthogonal, i.e. they can be applied at the same time independently without producing any interference. In addition, they will help each other in reducing the size of the mining problem. The data reduction operated by one constraint, i.e. the shortening of transaction
3.3. A new data-reduction framework for constrained pattern mining

Table 3.1: Classification of commonly used constraints.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Anti-monotone</th>
<th>Monotone</th>
<th>Succinct</th>
<th>Convertible</th>
<th>CLAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(X.A) ≥ v</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>min(X.A) ≤ v</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>max(X.A) ≥ v</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>max(X.A) ≤ v</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>count(S) ≤ v</td>
<td>yes</td>
<td>no</td>
<td>weakly</td>
<td>anti-monotone</td>
<td>yes</td>
</tr>
<tr>
<td>count(S) ≥ v</td>
<td>no</td>
<td>yes</td>
<td>weakly</td>
<td>monotone</td>
<td>no</td>
</tr>
<tr>
<td>sum(X.A) ≤ v</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>anti-monotone</td>
<td>yes</td>
</tr>
<tr>
<td>sum(X.A) ≥ v</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>monotone</td>
<td>no</td>
</tr>
<tr>
<td>range(X.A) ≤ v</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>range(X.A) ≥ v</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>avg(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>avg(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>median(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>median(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>strongly</td>
<td>yes</td>
</tr>
<tr>
<td>var(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>var(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>std(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>std(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>var_N-1(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>var_N-1(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>md(X.A) ≥ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>md(X.A) ≤ v</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

For the sum constraint we assume that every attribute value is non negative.

or even their removal, may introduce new pruning chances for other constraints regardless whether they operate on the search space, as we have shown with the \(C_{AM}-C_{M}\) trade-off, or the operate on the data.

Supported by these consideration, we designed the mining engine of CONQUEST as a level-wise bottom-up data-reduction boosted algorithm for constrained pattern mining. In out previous works [9] [10] we provided data-reduction algorithms for every class of constraints. These algorithms showed to perform in most cases better than other FP-tree based ones. Thanks to the independence of each strategy, we can now provide a comprehensive algorithm dealing with any possible conjunction of constraints.

In the following we describe first the core FIM algorithm that we extended to deal with constraint, and then we describe the pseudo-code of CONQUEST’s mining engine.
### 3. ConQueSt: constraint-based pattern discovery

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Algorithms</th>
<th>Search Space</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-Monotone $C_{AM}$</td>
<td><strong>Apriori</strong> 1</td>
<td>Discard supersets of infrequent itemsets.</td>
<td>Remove useless items in each transaction.</td>
</tr>
<tr>
<td>Monotone $C_M$</td>
<td><strong>ExANTE</strong> 8</td>
<td>Implicitly from the data reduction.</td>
<td>Repeatedly remove invalid transactions and singletons that become infrequent.</td>
</tr>
<tr>
<td></td>
<td><strong>ExAMiner</strong> 7</td>
<td>Implicitly from the data reduction.</td>
<td>Remove invalid transactions</td>
</tr>
<tr>
<td></td>
<td><strong>Dual-Miner</strong> 12</td>
<td>Top Down visit</td>
<td>-- -- --</td>
</tr>
<tr>
<td>Succinct Anti-monotone $C_{AMS}$</td>
<td><strong>CAP</strong> 40</td>
<td>Implicitly from the data reduction.</td>
<td>Remove all itemsets that do not satisfy the constraint from the database.</td>
</tr>
<tr>
<td>Succinct Monotone $C_{MS}$</td>
<td><strong>CAP</strong> 40</td>
<td>Remove invalid itemsets that are not subset of a valid one.</td>
<td>-- -- --</td>
</tr>
<tr>
<td></td>
<td><strong>ExAMiner</strong> 7</td>
<td>Implicitly from the data reduction.</td>
<td>The same of a monotone constraint.</td>
</tr>
<tr>
<td>Convertible $C_{CAM}, C_{CM}$</td>
<td><strong>Fic-A, Fic-M</strong> 56</td>
<td>Bottom-Up Depth-first with item reordering</td>
<td>-- -- --</td>
</tr>
<tr>
<td></td>
<td><strong>ExAMiner-Lam</strong> 9</td>
<td>Implicitly from the data reduction.</td>
<td>Remove useless items from transactions. Exploit loose anti-monotonicity.</td>
</tr>
<tr>
<td></td>
<td><strong>ExAMiner-Lam</strong> 9</td>
<td>Implicitly from the data reduction.</td>
<td>Remove transactions that do not contain any valid itemset.</td>
</tr>
</tbody>
</table>

Table 3.2: A summary of search space and data reduction strategies for the different classes of constraints.

#### 3.3.1 The basic mining engine

The ConQueSt mining engine is based on DCI 52, a state-of-the-art high-performance frequent itemsets mining algorithm.

DCI explores the search space level-wise, like Apriori, by first discovering the frequent itemsets of length one, then the frequent itemsets of length two, and so on until no longer frequent itemset exists. As other Apriori-like algorithms, DCI reiterates two basic steps. Given a collection of frequent itemsets of length $k$, a
new set of possibly frequent itemsets of length $k + 1$ is produced via the candidate generation function. Then, their actual support is calculated by scanning the dataset via the support counting function. These two steps are repeated until all the frequent itemsets have been discovered.

We chose a level-wise algorithm because it best fits our data-reduction framework. At each iteration, each active constraint may process each transaction of the dataset and apply its data-reduction opportunities. Note that the items removed by one constraint, may increase the chance that another constraint will prune a significant portion of the transaction.

Although the depth-first visit is now considered a more fashionable strategy, DCI is as fast as other algorithms thanks to its internal vertical bitmap representation of the dataset, and the associated tid-list intersection counting method, based on fast bitwise operations.

The added value that really makes DCI different from other algorithms is its nice feature of being resource and data aware.

- it is resource aware because, unlike other algorithms, it performs the first iterations out-of-core, and at each step prunes useless information from the original dataset thus reducing the amount of data to be used in the following iterations. When the reduced dataset is small enough to be loaded in main memory, it is converted and stored in-core as a vertical bitmap. The compact vertical representation allows a fruitful use of CPU’s cache due to the spatial and temporal locality in accessing data.

- it is data aware because its behavior changes in presence of sparse or dense datasets. It uses an ad hoc representation (similar to the run length encoding) in the case of sparse datasets, and detects highly correlated items to save bitwise works in the case of dense datasets.

Being resource and data aware is not only a nice feature of DCI, but it is also a strong requirement of ConQuest, due to the need of quickly mining real world datasets. This is the reason of our choice of DCI as the main building block of our mining engine. ConQuest, by inheriting the same characteristics as DCI, turns out to be an extremely robust and fast software.

3.3.2 The ConQuest algorithm

The pseudo-code of ConQuest mining engine is given in Algorithm 3. Notice how this framework fits and enhances each single algorithmic contribution participating to the mining engine.

Before the main loop of the algorithm starts, all the data-reduction opportunities are exploited in order to have from the beginning a very small dataset (line 5).

Then the Horizontal Loop starts (lines 6-12). The horizontal_count procedure reads the disk resident dataset and calculates the support of the candidate itemsets.
Algorithm 3 ConQuest mining engine

1: function ConQuest \((\mathcal{D}, \overline{\sigma}, C)\)  \(\triangleright\) where \(C\) is the set of constraints
2: \(k \leftarrow 1\)
3: \(\mathcal{L}_k \leftarrow \text{find-frequent-singletons}(\mathcal{D}, \overline{\sigma})\)
4: \(\mathcal{R}_k \leftarrow \{X \in \mathcal{L}_k \mid C(X)\}\)  \(\triangleright\) valid itemsets
5: \(\mathcal{D}_k \leftarrow \text{data-reduction}(\mathcal{D}, C)\)  \(\triangleright\) remove useless items and transactions
6: while \(\mathcal{D}_k\) doesn’t fit in main memory AND \(\mathcal{L}_k \neq \emptyset\) do  \(\triangleright\) Horizontal Loop
7: \(k \leftarrow k + 1\)
8: \(\mathcal{C}_k \leftarrow \text{candidate-generation}(\mathcal{L}_{k-1}, C)\)
9: \(\langle \mathcal{L}_k, \mathcal{D}_k \rangle \leftarrow \text{horizontal-support-counting}(\mathcal{D}_{k-1}, \mathcal{C}_k, \overline{\sigma}, C)\)
10: \(\mathcal{R}_k \leftarrow \{X \in \mathcal{L}_k \mid C(X)\}\)
11: \(\mathcal{D}_k \leftarrow \text{data-reduction}(\mathcal{D}_{k-1}, C)\)  \(\triangleright\) remove useless items and transactions
12: end while
13: \(BM_k = \text{vertical-format}(\mathcal{D}_k)\)  \(\triangleright\) Vertical Loop
14: while \(\mathcal{L}_k \neq \emptyset\) do
15: \(k \leftarrow k + 1\)
16: \(\mathcal{C}_k \leftarrow \text{candidate-generation}(\mathcal{L}_{k-1}, C)\)
17: \(\langle \mathcal{L}_k, BM_k \rangle \leftarrow \text{vertical-support-counting}(BM_{k-1}, \mathcal{C}_k, \overline{\sigma}, C)\)
18: \(\mathcal{R}_k \leftarrow \{X \in \mathcal{L}_k \mid C(X)\}\)
19: end while
20: return \(\bigcup_{i=1}^{k} \mathcal{R}_i\)  \(\triangleright\) the collection of valid itemsets
21: end function

\(\mathcal{C}_{k+1}\) in order to produce the set of frequent itemsets \(\mathcal{L}_{k+1}\). Also, data reduction properties are exploited in order to remove useless items and transactions, thus producing a new reduced dataset \(\mathcal{D}_{k+1}\) to use during the subsequent iteration.

At every iteration the algorithm will work on a smaller and smaller amount of data. When the dataset is small enough, it will be entirely stored in the main memory by using a vertical bitmap. The bitmap format does not allow to prune the dataset further during the Vertical Loop (lines 13-19). Nonetheless, the compact memory resident image of the dataset will enhance the temporal and spatial locality of DCI, which can complete the mining with its efficient use of CPU cache without the need of additional data-reduction.

This framework can be easily extended to handle future user-defined constraints by exploiting their data-reduction properties in a similar way.
3.4 Experimental evaluation

In our experiments we used four different datasets, namely Retail, Accidents, WebDocs and USCensus1990, whose characteristics are described in detail in Appendix A. The first is a sparse dataset with short frequent itemsets, while the second is still sparse but with many frequent long itemsets. The remaining two are quite large and in a way more complex.

In evaluating a pattern mining system it is very important to test different datasets because each of them raises different difficulties for a pattern mining algorithm. Sparse and small datasets are usually very easy to mine, since the overall number of extracted frequent itemsets is not large. On the other hand, dense datasets produce a much larger number of frequent itemsets, and since they are usually characterized by having long transactions, it is usually difficult to effectively apply frequency-based or other pruning techniques to them. Additionally, the two large datasets force the algorithm to explore a huge number of candidates, thus increasing its memory requirements.

Since these datasets are pure transactional datasets, items have no associated attribute. We thus generated a synthetic attribute for each distinct item by using random values drawn from a uniform distribution in the interval $[0, 1000]$. The aim of our performance experiments was to asses the pruning power of our data-reduction techniques, to show the absolute effectiveness of the system compared with other specialized constrained frequent pattern mining algorithm, and, finally, to show the goodness coming from the synergy of a powerful data-reduction approach with a fast mining algorithm.

Dataset pruning power

We tested CONQUEST pruning power on the first three datasets by using three different constraints. We forced the algorithm to perform the horizontal setup only, i.e. the only one where data-reduction is performed. During each iteration the algorithm produces a reduced dataset, whose size is plotted in order to evaluate the effectiveness of the different pruning strategies.

The constraints we tested were $\text{sum}(S.A) \geq v$ (hereinafter denoted $\text{sumgeq}$ for simplicity) that is monotone, $\text{avg}(S.A) \geq v$ (avggeq) that is convertible, and $\text{var}(S.A) \leq v$ (varleq) that is loose-antimonotone.

Figure 3.2 and Figure 3.3 show the results of these tests, where we varied the threshold values of each constraint. For each set of tests we also report the baseline case, where all the frequent itemsets satisfy the constraint (e.g., $\text{sum}(X.A) \geq 0$). In this way we can compare our data-reduction techniques with the sole anti-monotone frequency based pruning.

With the Retail dataset, the avggeq and varleq were effective only for very selective thresholds, and only starting from the fourth iteration. However they will soon prune a large number of transactions almost emptying the dataset in a few
Figure 3.2: Effectiveness of the various pruning strategies. We used the following absolute minimum support thresholds: $\sigma = 3$ for Retail and $\sigma = 100000$ for Accidents.
Figure 3.3: Effectiveness of the various pruning strategies. We used the following absolute a minimum support thresholds of $\sigma = 180000$ in WebDocs dataset.
successive iterations. The `sumgeq` constraint requires more iterations to reduce significantly the dataset, but it immediately removes a large part of transactions starting from the very first iterations. Moreover, its behaviour is more consistent with respect to the selectivity of the constraint.

Since `Accidents` is a more dense dataset, the pruning here is more difficult. In fact, the frequency-based pruning is not able to remove a valuable number of transactions during the first ten iterations. Even the pruning given by the `sumgeq` constraint is poor for every thresholds used, because it works at a transaction granularity: it removes the whole transaction or does nothing. Conversely, the other two constraints perform much better. The advanced pruning of `avggeq` works at item granularity, and allows to remove many items from each transaction, in turn increasing chances for the removal of the whole transaction. The last constraint `varleq`, behaves pretty well because of its quasi-antimonotone properties: computation may actually end after a few iterations.

In the `WebDocs` dataset, the frequency-based anti-monotone pruning works better than with the `Accidents` dataset, but is not effective as with the `Retail` one. This means that, while being generally sparse, there are many dense portion of the dataset. As a result the monotone pruning helps in mining the `sumgeq` constraint by removing a small, but significant number of transactions, just from the first iterations. The other two constraints work much better and consistently for every threshold used, even if they are not able to perform any pruning in first two iterations.

As expected, monotone pruning is more effective when the dataset is sufficiently sparse, i.e. when a lot of short transactions can be easily pruned. On the other hand it allows to remove transactions from the very beginning of the algorithm. The other two pruning strategies need instead a few iterations to make their contribution evident. However, after such initial stage, they can immediately remove a large portion of transactions, often allowing the algorithm to end some iterations in advance due to the emptied dataset at that point.

**Comparison with other algorithms**

In Figure 3.4 we compare the performance of `ConQueST` with other two state-of-the-art algorithms `Fic-A` and `Fic-M`. We used the same three datasets discussed above, and the convertible constraint `avggeq`. In this test we wanted to stress all the algorithms, and for this reason we used low supports and highly selective constraint thresholds.

As expected, `Fic-M` was never able to compete with `Fic-A`, since, as we discussed above, it does not perform an actual pruning of the search space. Note the bars of Figure 3.4, where an exclamation mark appears: in these cases we artificially interrupted `Fic-A` or `Fic-M` after hours of computation, or that the algorithm aborted.

In all the tests with all the three datasets `ConQueST` was the clear winner.
Figure 3.4: CONQUEST versus other algorithms. We used the following absolute minimum support thresholds: $\sigma = 3$ for Retail, $\sigma = 100000$ for Accidents and $\sigma = 180000$ for WebDocs dataset. The exclamation mark means that the algorithm aborted before completion.
Even if it was designed to handle several classes of constraints at the same time, still it resulted to be much faster than specialized algorithms such as Fic-A. Great part of this effectiveness has to be credited to the mining core coming from DCI.

Finally, in mining the largest dataset among the three (WebDocs) ConQueST was the only one able to run to completion. Both the other two algorithms terminated because of segmentation fault errors.

**Synergy of the two approaches**

In the previous two sections we showed the effectiveness of the data reduction process, and the high performance of the pattern mining core. Notice that dataset pruning is possible only during the horizontal iterations, while it is not applicable during vertical iterations.

In this section we will show the synergy of the combination of these two approaches. A few first horizontal iterations will speed up the vertical phase of the algorithm.

For this test we used the USCensus1990 dataset and the avggeq constraint. Even if this dataset is smaller than webdocs, it has a larger number of transactions. Additionally, if we consider that the average transaction length (68) is large compared to the total number of items (396), it is clear that this dataset is very dense. In other words, we chose a large dataset so that to have a large number of possible frequent itemsets. Moreover, since the dataset is dense, we can expect that this feature makes the pruning process very complex.

We run ConQueST by varying the number of initial horizontal iterations from 2 to 6, and the obtained results are shown in Figure 3.5. The rationale was to find out the (possibly small) number of horizontal iterations that improves the overall performance of the mining process.

Consider the constraint avggeq with threshold 700: Figure 3.5(a) shows that the running time of the algorithm increases dramatically when increasing the number of horizontal iterations. The reason of this behavior is shown in Figure 3.5(b), which reports the number of candidate itemsets explored by the algorithm for different number of horizontal iterations performed. For avggeq with threshold 700, this number does not change significantly for the various iterations. In this case, the horizontal iterations are not able to introduce a significant pruning, and, being slower than the vertical ones, they only degrade the performance.

When increasing the selectivity of the constraint to avggeq with threshold 750, the running time decreases significantly due to the reduced size of the dataset, that also reduces by two or more orders of magnitude the number of candidate itemsets. It is clear that the best choice is to start the vertical iterations from the fourth pass, since the cost of additional horizontal iterations is not worth the dataset reduction provided.

The conclusion is that the horizontal technique is not efficient enough to bear the mining effort, but it can highly reduce the cost of the following phases of the mining
Figure 3.5: Effectiveness of the combined horizontal and vertical approach on census dataset with a minimum support threshold $\sigma = 1500000$.

process. However the combination of the two approaches results to be definitely successful.

### 3.5 The ConQuest system

ConQuest has actually become an association rule mining software. Its modular architecture reflects the aim of realizing a complete KDD process, even if limited to
the mining of constrained frequent itemsets. The main building blocks of the system are the following:

- The **Graphical User Interface (GUI)** is responsible for every interaction between the user and the other modules. This advanced GUI allows the user to navigate through the tables of a relational database and to define complex mining queries via mouse-clicks. It is possible to select a subset of the database, to discretize fields, to define items, attributes and transaction, and finally to set constraints. Everything is translated into an enriched SQL query that will be interpreted in the next step. The GUI can also provide some visualization of the results of the mining task in form of a table. It also has some convenient interface that allows the user to apply some accurate tuning to the parameters of the mining queries. In Figure 3.7 we show a screenshot of ConQueSt.

- The **Query Interpreter (QI)** has the main task of transforming relational tables into a transactional dataset. Moreover, since any interaction with the underlying DBMS is caused by the user-provided queries that are processed by the QI, it is also in charge of other optimization tasks, such as caching, to improve the overall system usability.

- The core of the pattern extraction process is performed by the **Mining Engine (ME)**, which encloses the state of the art of high performance and constraint-based pattern mining algorithms in a novel unifying framework discussed in Section 3.3.

Figure 3.6 depicts the main architectural blocks of the architecture of ConQueSt. While the GUI and the QI were developed in Java, the ME was developed in C++ for efficiency reasons, and wrapped in Java to allow the rest of the software package to invoke its methods. Note that the three main blocks interacts with the DBMS through a JDBC abstraction layer. Using the JDBC interface, the current version of the ConQueSt prototype can exchange data with PostgreSQL, Microsoft
Access, Oracle, MySQL, and SQL Server. Additional DBMSs can be interfaced by adding the suitable JDBC plug-in.

This subdivision in three distinct blocks reflects the main functionalities of our tool. It provides a better maintainability and extensibility of the software, and even a smart deployment when required. In fact, as a proof of the good separation between independent tasks, the various modules could be disjoint software packages, running on different machines, and cooperating through Java RMI. For instance, the GUI may run on the user machine, while the query interpreter may be located in a different site where a fast access to the database is provided, and finally the mining engine may be run on a high performance cluster serving many users with many different tasks. Finally, note that the JDBC layer allows us to ignore the physical location of the DBMS server.

We hope to make this software publicy available very soon.

3.6 Conclusions

We designed CONQUEST, the first algorithm able to deal with any conjunction of constraints. It is a level-wise algorithm were each iteration is accompanied with a reduced version of the original dataset. The algorithm relies on the data reduction opportunities given by each constraint. In presence of multiple constraints, every data reduction can be applied independently in any order. In fact, the pruning exploited by one constraint can only increase the reduction opportunities of another constraint.

Thanks to our data-reduction technique we can deal with all the high performance issues we have decided to take into consideration.

Computational complexity. We showed that each class of constraints provides many data reduction opportunities. It is in fact possible to remove items from transactions or even whole transactions before starting a new iteration. Depending on the constraint, we in fact provided several criteria to decide whether or not an item or a transaction will support a valid and frequent itemset in the subsequent iterations.

By pruning some transactions and reducing the length of the remaining ones, at each iteration, we considerably speed up the support counting phase.

Data size. CONQUEST has an adaptive behaviour according to the size of the dataset. If the dataset is too large to be loaded in main memory, it is left on disk and stored in horizontal format. Iteration by iteration the dataset gets smaller and smaller, and only when it can fit in main memory it is loaded and stored in a vertical format that allows faster support computations via tid-lists intersections.

Regardless the size of the database, the algorithm can successfully complete the mining task without running out of memory and without causing memory swaps.
3.6. Conclusions

Data dimensionality. In the context of constrained pattern mining we assume the user to adopt low minimum support thresholds, thus increasing the size of the search space. Some constraint may allow to prune the collection of candidates in a similar way as the apriori principle, but unfortunately there is not a strategy significantly effective.

What is peculiar to our algorithm, is that the data reduction exploited on the dataset has the positive side effect of pruning also the search space. In fact, by pruning transactions, we decrease the support of invalid itemsets. When an invalid itemset becomes infrequent, then all of its superset will be disregarded because of the powerful apriori principle.

We can thus overcome the dimensionality problem, since the algorithms reduces significantly the number of candidate itemsets.

As a result, CONQUEST can deal with more than one constraint at the same time. Also, it showed to outperform the other specialized algorithms and to be successful where other algorithm fail to run to completion.

Conquest is now a young software that still needs many improvements, but it can accompany the user along all the steps of a successful knowledge discovery process. A demo of this software was presented at the ICDE 2006 International Conference of Data Engineering [6].

By using our experience in constrained frequent pattern mining, we wanted to highlight how much high performance data mining is important. The issues we have described in the previous chapter show up behind every data mining tasks, regardless the type of pattern and the type of data. To design efficient and robust algorithm is always mandatory, and to achieve this goal high performance issue must be taken into account.
3. **CONQUEST**: constraint-based pattern discovery
4

DCI-Closed: a new algorithm for mining closed frequent itemsets

In this section we will present a new scalable algorithm for discovering closed frequent itemsets, a lossless and condensed representation of all the frequent itemsets that can be mined from a transactional database.

We introduce a new algorithm called DCI-Closed, and compare it with the other state-of-the-art algorithms. Thanks to its duplicate detection technique and to its internal representation of the datasets, it outperforms other state-of-the-art algorithms, and more importantly, it is a robust and scalable algorithm.

4.1 Motivations for closed frequent itemsets

In many cases it is useful to use low minimum support thresholds $\sigma$. But, unfortunately, the number of extracted patterns grows exponentially as we decrease $\sigma$. It thus happen that the collection of discovered patterns is so large to require an additional mining process that should filter the really interesting patterns.

The same holds with dense datasets, such as census data. These contain strongly correlated items and long frequent patterns. In fact, such datasets are hard to mine even with high minimum support threshold. The Apriori property does not provide an effective pruning of candidates: every subset of a candidate is likely to be frequent.

In conclusion, the complexity of the mining task becomes rapidly intractable by using conventional algorithms.

Closed itemsets are a solution to the problems described above. These are obtained by partitioning the lattice of frequent itemsets into equivalence classes according to the following property: two distinct itemsets belong the the same class if and only if they occur in the same set of transactions. Closed itemsets are the collection of maximal itemsets of these equivalence classes.

When a dataset is dense, the number of closed itemsets extracted is order of magnitudes smaller than the number of frequent ones. This leverages the problem of the analyst of analyzing a large collection of patterns. Also, they reduce the complexity of the problem, since only a reduced search space has to be visited.
High performance data mining often tries to solve an expensive problem looking for an equivalent one that it is easier to solve. In fact, from closed itemsets it is trivial to generate the whole collection of frequent itemsets along with their supports. In other words, frequent and closed frequent itemsets are two different representations of the same knowledge. Moreover, recent FIM algorithms, use the concept of closed itemsets to speed up their computation, and when possible they explicitly extract closed itemsets and then generate frequent ones in a sort of post-processing phase. The first of these kind of algorithms was PASCAL [5], and now any FIM algorithm uses a similar expedient.

More importantly, association rules extracted from closed itemsets have been proved to be more meaningful for analysts, because many redundancies are discarded [4]. Suppose to have two frequent rules \( r_1 : \{ \text{diapers} \} \rightarrow \{ \text{milk, beer} \} \) and \( r_2 : \{ \text{diapers} \} \rightarrow \{ \text{milk} \} \) having the same support and confidence. In this case, the rules \( r_1 \) is more informative since it includes \( r_2 \): it tells something more about the implications of item diapers. Note that \( \sigma(\text{diapers, milk}) = \sigma(\text{diapers, milk, beer}) \), i.e. the two itemsets occur in the same set of transactions and therefore they belong to the same equivalence class, but since \( r_2 \subset r_1 \) then \( \{ \text{diapers, milk} \} \) is not closed. Thus, an algorithm based on closed itemsets will not generate the redundant rule \( r_2 \).

This is why many algorithms for mining closed frequent itemsets have been proposed, and why the idea of closed itemsets has been borrowed by other frequent pattern mining tasks: there are algorithm for the extraction of closed sequences[71], closed trees[16], closed graphs[70], etc.

The idea of closed itemsets come from the application of formal concept analysis. This was formalized in the early 80s by Rudolf Wille [67] and years later it has found many application in data mining, information retrieval and artificial intelligence.

4.2 The closed frequent itemsets mining problem

We now introduce formally the problem of mining closed frequent itemsets. Let \( T \) be a set of transaction in \( D \), and \( I \) an itemset, the concept of closed itemset is based on the two following functions \( f \) and \( g \):

\[
\begin{align*}
  f(T) &= \{ i \in I \mid \forall t \in T, i \in t \} \\
  g(I) &= \{ t \in D \mid \forall i \in I, i \in t \}
\end{align*}
\]

Function \( f \) returns the set of items included in all the transactions in the set \( T \), i.e. their intersection, while function \( g \) returns the set of transactions supporting a given itemset \( I \). As an exercise, we could write \( \sigma(I) = |g(I)| \).

**Definition 4.1** An itemset \( I \) is said to be closed if and only if

\[
c(I) = f(g(I)) = f \circ g(I) = I
\]

where the composite function \( c = f \circ g \) is called Galois operator or closure operator.
The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class if and only if they have the same closure. Equivalently, two itemsets belong to the same class iff they are supported by the same set of transactions. We call these partitions of the lattice closure based equivalence classes.

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

**Example 4.1** Figure 4.1 shows the lattice of frequent itemsets derived from the simple dataset reported in the same figure, mined with \( \sigma = 1 \). We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that closed itemsets (six) are remarkably less than frequent itemsets (sixteen).

**Definition 4.2 (CFIM: Closed Frequent Itemset Mining)**

Let \( \mathcal{D} \) be a transactional dataset and \( \sigma \) a given minimum support threshold, the
Frequent Closed Itemset Mining Problem requires to discover all the itemsets \( X \) such that \( \sigma(X) \geq \overline{\sigma} \) (they are frequent) and \( c(X) = X \) (they are closed).

Other concise representations \([50]\) of frequent itemsets where proposed, such as \( \delta \)-free sets \([11]\), disjunction-free sets \([13, 38]\), generalized disjunction-free sets \([39]\), non-derivable itemsets \([14]\), and so on.

The \( \delta \)-free sets introduce a toleration threshold \( \delta \) in the concept of closed itemsets: \( X \) is a \( \delta \)-free set if no superset having support at least \( \sigma(X) - \delta \) exists. A closed itemset is a 0-free set. Disjunction-free sets allow to introduce one (and only one) disjunction in an itemset, e.g. \( \{a \land b \lor c\} \), in contrast with the usual definition of a itemset where every item is required to be present in an occurrence, i.e. \( \{a \land b \land c\} \).

Similarly, generalized disjunction-free sets explore the case were an undefined number of disjunctions are allowed. Finally, non derivable itemsets, use Bonferroni-type inequalities \([20]\) to derive the support of a subset of frequent itemsets, whose support cannot be derived and thus called non derivable itemsets.

However, closed itemsets are still the most popular condensed representation. It does not have some drawbacks of other disjunctive sets that need to maintain some infrequent itemsets, and they are not costly to extract as non derivable itemsets. More than that, closed itemsets are the easiest to understand for an analyst. For the remainder of this these we focus on closed itemsets and related algorithmic issues.

4.3 The state-of-the-art of closed frequent itemsets mining

In this section we review some significant closed frequent itemsets mining algorithms, and at the same time we deeply analyze the major issues of this pattern extraction problem. These algorithms are very similar to FIM ones, from which they borrow their computational skeleton. Indeed, we describe both level-wise and depth-first algorithms.

A completely new problem accompanies CFIM algorithms, that we call duplicate detection problem or spurious itemsets removal. This is related with recursive dataset projections exploited by depth-first algorithms. We formally analyze this problem and compare the solutions provided by various algorithms. We propose with our algorithm a new efficient duplicate detection technique, that, in turn, allows to subdivide the original mining problem into independent sub-tasks. This will be an important point also for the following chapters.

4.3.1 A level-wise algorithm

CFIM algorithms share the same backbone with FIM algorithms. They are based on the same visiting strategy of the search space, either level-wise or depth-first, but with some additional steps for calculating closures.
4.3. The state-of-the-art of closed frequent itemsets mining

A-Close \[55\]

The first proposed algorithm for mining closed itemsets was A-Close. This was designed on the solid base of Apriori. The idea is to first generate a small set of frequent itemsets, called generators, and then, by calculating their closures, to derive the complete collection of closed itemsets. The resulting algorithm is obviously much faster than Apriori since the number of generators is significantly smaller than the number of frequent itemsets.

The authors chose as generators the set of key-patterns since they have an Apriori-like property that helps the mining:

Property 4.1 (Key-Patterns Downward Closure)

An itemset \(X\) of cardinality \(k\) is a key-pattern if and only if none of its subsets of cardinality \(k-1\) has the same support of \(X\).

A slightly modified Apriori extracts all the key-patterns during the first step of the algorithm, and their closures are computed in a second step. The correctness of the algorithm is given by the fact that each closure based equivalence class has at least one key pattern.

Unfortunately, there is a great drawback of using key-patterns. It is easy to see these are the minimal itemsets of closure based equivalence classes. Indeed, each equivalence class has only one closed itemset but it may have several key-patterns. Therefore, A-CLOSE, during its second step, may extract the same closed itemset more than once. This introduces a significant overhead.

Example 4.2 The closed itemset \(\{abcd\}\) of Figure 4.1 would be mined twice, since it can be obtained as the closure of two minimal elements of its equivalence class, namely \(\{ab\}\) and \(\{bc\}\).

4.3.2 First generation depth-first algorithms

As for FIM algorithms, also in this case a depth-first approach performs generally better than a level-wise algorithm. This is what the same authors of ECLAT and FP-GROWTH showed with their CFIM algorithms Charm \[75\] and Closet \[57\].

These depth-first algorithms use again the idea of finding a set of generators and then calculating their closure. Differently from A-CLOSE, the closure is computed in a sort of in-line fashion. We call this technique closure climbing. Rather than computing closure after the whole collection of generators has been extracted, they compute the closure of a generator as soon as this is found, and then they use the resulting closed itemset to produce new generators.

The visit of the search space is indeed very similar to FIM algorithm, but with interleaved closure computations. Given a closed itemsets \(X\), new supersets of length \(|X| + 1\) are built and used as generators. Their closure is computed and then used to reiterate the process.
The closure computation involves jumps in the lattice of frequent itemsets, which may or may not violate the underlying lexicographical order. Hence, similarly to the A-CLOSE approach based on key patterns, it is possible to generate the same closed itemset multiple times.

Example 4.3 As shown in Figure 4.1, the itemset \{a\} is a generator since it can be obtained as a superset of the closed itemsets \emptyset, and its closure is \{acd\}. But also \{cd\} is a generator obtained as a superset of \{c\}, and it has the same closure \{acd\}. In conclusion, the closure of the itemset \{cd\} involves a backward jump to the itemset \{acd\} \prec \{cd\}, thus producing again the same closed itemset.

It thus becomes mandatory to solve the duplicate detection problem, that is to understand whether or not a generator may lead to an already discovered closed itemset. We address with the label first generation those algorithms that use the following Subsumption Lemma to detect a duplicate.

**Lemma 4.1 (Subsumption Lemma)**

Given an itemsets \(X\) and closed itemset \(Y = c(Y)\), if \(X \subset Y\) and \(\sigma(X) = \sigma(Y)\) then \(c(X) = Y\). In this case we say that \(X\) is subsumed by \(Y\).

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

As soon as a closed itemset is discovered, it is stored in an incremental data structure. This historical collection is used to detect duplicates. Before calculating the closure of a new generator \(X\), thus saving some computation, we can check whether there exist a closed itemset \(Y\) that includes \(X\) and that has the same support of \(X\). In this case, it holds that \(c(X) = Y\) and therefore the itemset \(X\) would lead to a duplicate.

Example 4.4 The generator \{cd\} would be promptly detected as leading to a duplicate because it is a subset of \{acd\} which has identical support. The itemset \{acd\} was already mined according to the lexicographical order and stored into the historical collection.

Another important aspect of this kind of algorithms must be taken into account to guarantee their correctness. The great advantage of using dataset projections to speed up the support computation, has a negative side-effect when calculating closures.

Recall that the correctness and completeness of a FIM algorithm are based on a lexicographical visit of the search space. According to this, the algorithm assumes that if \(X\) is the current node then only itemsets \(Y \mid X \prec Y\) can be visited afterward. This assumption allows an aggressive pruning of the projected datasets. First, all the elements of \(X\) can be removed, simply because we know that \(X\) occurs in every transaction of \(D_X\) by construction. Then, not only it is possible to remove all those
transactions that do not contain \( X \), but also all those items that cannot support the descendants of \( X \), i.e. all those items such that \( X \not\supseteq \{X \cup i\} \). Simply stated, given the sorted itemset \( X = \langle x_1 \ldots x_k \rangle \), then all the items \( i \preceq x_k \) are removed.

This last pruning is crucial because it removes some items that could belong to \( X \)'s closure. While this pruning leaves intact any information about the itemsets \( Y \mid X \prec Y \), it is not true that \( X \prec c(X) \), meaning that the pruning may affect the occurrences of \( c(X) \) in the projected dataset. Therefore, in general, it is not possible to calculate the closure of \( X \) using only the projected dataset \( D_X \).

Let’s consider the following Lemma that is commonly used to compute closures.

**Lemma 4.2 (Extension Lemma)**

Given an itemset \( X \) and an item \( i \in I \), if the set of transactions supporting \( X \) is a subset of the transactions supporting \( i \), then \( i \) belongs to the closure of \( X \) and viceversa, i.e. \( g(X) \subseteq g(i) \iff i \in c(X) \).

**Proof.**

\((g(X) \subseteq g(i) \Rightarrow i \in c(X))\):

Since \( g(X \cup i) = g(X) \cap g(i) \), \( g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X) \). Therefore, if \( g(X \cup i) = g(X) \) then \( f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X) \).

\((i \in c(X) \Rightarrow g(X) \subseteq g(i))\):

If \( i \in c(X) \), then \( g(X) = g(X \cup i) \). Since \( g(X \cup i) = g(X) \cap g(i) \), \( g(X) \cap g(i) = g(X) \) holds too. Thus, we can deduce that \( g(X) \subseteq g(i) \).

The above Lemma is used to calculate the closure of a generator \( X \) by discovering, one by one, all the items belonging to \( c(X) \). The exploitation of the Extension Lemma is trivial when the projected dataset is available. Given an itemset \( X \) and its conditional dataset \( D_X \), an item \( i \) belongs to the closure of \( X \) if and only if its support in the projected dataset \( \sigma_{D_X}(i) \) is equal to number of transactions \( D_X \). In fact, since \( D_X \) contains exactly those transactions that are superset of \( X \), the item \( i \) may occur in every transaction of \( D(X) \) if and only if \( g(X) \subseteq g(i) \), which implies \( i \in c(X) \).

**Example 4.5** Consider the generator \( a \), by applying the closure operator \( c(\cdot) \), we would find that its closure is \( \{acd\} \). If we built the projected dataset \( D_a \), i.e. the set of transactions containing \( a \), this would include only the second and third transactions of \( D \) but pruned from \( a \) itself: \( D_a = \{bcd, cd\} \). Note that the items \( c \) and \( d \) occur in every transaction of \( D_a \), meaning that \( a \) occurs always together with \( c \) and \( d \), and therefore \( \{cd\} \subset c(a) \).
Unfortunately, as we highlighted above, since \( X \not\subset c(X) \), the projected dataset \( D_X \) does not contain all the information, i.e. items, needed to compute correctly the closure \( c(X) \). In general, a projected dataset allows to compute only an incomplete or partial closure of \( X \), we call it a *spurious closed itemset*. According to the aforementioned projecting strategy, this partial closure will contain those items \( i \in c(X) \) such that \( x_k \prec i \), and not those items \( j \in c(X) \) such that \( j \prec x_k \). We will deal again with the problem of spurious itemsets in Chapter 5.

**Example 4.6** Considering the generator \( \{bc\} \), by construction \( D_{bc} = \{d\} \), which means that \( c(bc) = \{bcd\} \). By using \( D_{bc} \) only we would find that \( c(bc) = \{bcd\} \) which is a spurious itemset resulting from an incomplete closure computation. In fact the closure of \( \{bc\} \) is \( \{abcd\} \). We will deal again with the problem of spurious itemsets in Chapter 5.

Differently from the level-wise A-CLOSE algorithm, where multiple key-patterns may lead to the same (correct) closure at the price of scanning the whole dataset, by using projections, we risk to produce a set of spurious closed itemsets. This problem is crucial for what regards the order by which itemsets are discovered. Suppose that the closure of a generator cannot be computed in its projected database, if its correct closure was not already discovered and inserted in the historical collection, than the generator would not be detected as a duplicate and an incomplete closure, i.e. a spurious itemset, would be produced.

A long as a lexicographical order is used to traverse the search space, the correctness of depth-first algorithms can be guaranteed. If itemsets are not mined in such order, than there is no guarantee of correctness. This issue will be fundamental for us in Chapter 6 where we deal with a parallel mining algorithm.

**Example 4.7** Suppose the generator \( \{cd\} \) is discovered first, then its incomplete closure \( \{cd\} \) found in the dataset \( D_{cd} \) would be inserted in the historical collection as it was a correct closed itemsets.

**Charm** [75], **Closet** [57]

In Algorithm 4 we report the pseudo-code of the two depth-first CFIM algorithms Charm and Closet.

Given a frequent generator \( X \), they calculate the frequency of each single item occurring in the projected dataset \( D_X \). Those items having the same support as \( X \) belong to the closure \( c(X) \) of the generator. The resulting closed itemset is stored into the historical collection (lines 15–17).

New candidate generators \( Y = c(X) \cup i \) are built using those frequent items that were not found to be part of \( c(X) \). The algorithms first exploit the Subsumption Lemma to assess whether the new generator \( Y \) leads to a duplicate or not (line 20).

If \( Y \) does not lead to a duplicate, the corresponding projected dataset \( D_Y \) is built, and used to start a new recursion of the visit (line 22).
Note that a particular projection is applied (line 21). The items of the new closure \( c(X) \) are removed from \( D_X \), and then the resulting dataset is projected against the item \( i \) used to create the new generator.

Algorithm 4 CHARM and CLOSET: two depth-first CFIM algorithms.

1: function \( \text{Mine}(D, \sigma) \)
2: if \( |D| < \sigma \) then \( \triangleright \) the empty set is a special case
3: return \( \emptyset \)
4: end if
5: if this is CHARM then \( \triangleright \) store the dataset in main memory
6: \( D \leftarrow \text{vertical-format}(D) \)
7: else this is CLOSET
8: \( D \leftarrow \text{horizontal-format}(D) \)
9: end if
10: \( C \leftarrow \emptyset \) \( \triangleright \) the collection of frequent closed itemsets
11: \( \text{Mine-Node}(\emptyset, D, \sigma, C) \) \( \triangleright \) visit the lattice starting from \( \emptyset \)
12: return \( C \)
13: end function

14: procedure \( \text{Mine-Node}(X, D_X, \sigma, C) \)
15: \( I_X \leftarrow \text{get-frequent-singletons}(D_X, \sigma) \) \( \triangleright \) calculate the closure of \( X \)
16: \( c(X) \leftarrow X \cup \{ j \in I_X \mid \text{supp}_{D_X}(j) = \text{supp}_{D}(X) \} \)
17: \( C = C \cup c(X) \) \( \triangleright \) add to the collection of closed frequent itemsets
18: for \( i \in (I_X \setminus c(X)) \) do
19: \( Y \leftarrow c(X) \cup i \) \( \triangleright \) generate a new frequent generator
20: if \( \neg \exists Z \in C \mid Y \subset Z \land \sigma(Y) = \sigma(Z) \) then \( \triangleright \) duplicate detection
21: \( D_Y = \text{project}(D_X \setminus c(X), i) \) \( \triangleright \) build a new projection
22: \( \text{Mine-Node}(Y, D_Y, \sigma, C) \) \( \triangleright \) recursive mining
23: end if
24: end for
25: end procedure

Both CHARM and CLOSET inherit the same data structures and computing framework of their big brothers dEclat and FP-Growth respectively. They implement Algorithm [4], but they differ in the way the closed frequent itemsets are stored in order to exploit the Sub-sumption Lemma.

CHARM adopts a hash table, were the hash function is the sum of the transactions ids supporting an itemset. CLOSET uses a trie-like structure, indexed by a two-level hash. The first level is based on the last item of the itemset to be checked and the second on its support.
FP-Close [28]

FP-Close is inspired to Closet, thus using the same divide et impera approach and same FP-tree data structure. What makes FP-Close different from other CFIM algorithms is the application of the projecting approach to the historical collection of closed frequent itemsets. Not only a small dataset is associated to each node of the tree, but also a pruned subset of the closed itemsets mined so far is forged and used for duplicate detection. Indeed, this technique is called progressive focusing and it was introduced by [23] for mining maximal frequent itemsets. Together with other optimizations, this truly provides dramatic speed-up, making FP-Close order of magnitudes faster than Charm and Closet, and also making it worth to be celebrated as the fastest algorithm at the FIMI workshop 2003 [26].

4.3.3 A second generation of algorithms

Algorithm [4] maintains in main memory the collection of discovered closed itemsets during their entire execution. This turns out to have many drawbacks and to become very expensive when this collection become sufficiently large.

Such technique is expensive both in time and space. In time because it requires the possibly huge set of closed itemsets mined so far to be searched for the inclusion of each generator. In space, because, in order to efficiently perform set-inclusion checks, all the closed itemsets have to be kept in the main memory.

In a way, the need to maintain a global and evolving data structure does not comply with the general requirements of a divide et impera approach. Instead of having a pool of independent sub-tasks, a CFIM algorithm is decomposed in dependent sub-problems that actually cooperate and communicate, meaning that the output of one task, i.e. the extracted closed frequent itemsets, is the input of the next task. On the other hand, it is mandatory to exploit recursive projections in order to exhibit good performance.

We intend to address with the label “second generation” those algorithms that introduced alternative strategies that allow to detect non order-preserving generators overcoming the disadvantages described above, i.e. without maintaining an incremental collection of closed frequent itemsets. We include in this set of algorithms Closet+, described below, and DCI-Closed, which will be object of the next section.

Closet+ [66]

The Closet+ algorithm differs from Closet because it introduces a new dataset projection and a new duplicate checking technique, respectively called pseudo-projection and upward-checking.

A pseudo-projection consist of a set of pointers to portions of the transactions that are interesting relatively to an itemset X. Instead of rebuilding a new reduced
dataset, it handles a set of pointers to the transactions stored in the initially built FP-tree which records the whole dataset. Such FP-tree is thus re-used at every recursive step. A pseudo-projection can be viewed as a representation of $g(X)$.

The upward-checking, consists in the application of the closure operator $c$, according to which all the transactions $g(X)$, and thus all the paths in the FP-tree that contain itemset $X$, are intersected each other. If the resulting itemset is exactly $X$, i.e. $f(g(X)) = X$, then $X$ is closed, otherwise $X$ is a spurious closed itemset.

This approach avoids all the disadvantages of the previous algorithms, by sharing the whole dataset during the computation, and without maintaining recursive projected datasets or closed frequent itemsets. The advantage of sharing the whole dataset, is that no information loss occurs since no pruning takes place, and it is therefore possible to apply directly the closure operator.

However, the authors noticed that this approach is not convenient with every kind of dataset. We distinguish between dense and sparse datasets. Dense datasets have long transactions and contain strongly correlated items. Usually, very long frequent patterns can be extracted from dense datasets even if large support thresholds are used during the mining process. Conversely, datasets are defined as sparse when items are not mutually correlated, and transaction are quite short on average. Even when low support thresholds are employed during the mining process, the resulting frequent patterns extracted from sparse dataset are not long.

The authors of Closet+, use this new strategy only in sparse datasets. The rationale is that transactions are in this case generally short, and thus intersections can be performed quickly. In dense datasets, Closet+ adopts the same duplicate detection strategy of Closet. Note that it is exactly in the case of dense datasets that the mining of closed itemsets is really interesting. While in dense datasets closure based equivalence classes are large and the number of closed itemsets is order of magnitudes smaller than the number of frequent itemsets, in sparse datasets equivalence classes are small and the collection of closed frequent itemsets is not dissimilar from the one of frequent itemsets.

In conclusion, we must say that the FP-Close algorithm, which does not use this novel duplicate detection technique, is much faster than Closet+. However, in the next section we will see another strategy for removing spurious itemsets that can be used in dense datasets and that provides enhanced efficiency.

### 4.4 The DCI-Closed algorithm

Our proposed algorithm DCI-Closed [45] was named after DCI [52], a FIM algorithm from which it inherits the in-core vertical bit-wise representation of the dataset, and several optimization heuristics.

The first two passes of the algorithm are devoted to the construction of the internal representation of the dataset. A first scan is performed in order to discover the set of frequent singletons $L_1$. Then, a second scan is needed to create a vertical
bitmap of the dataset \( BM \). For each item \( i \in L_1 \), a 0-1 array of bits is stored, where the \( j \)-th bit is set if the item \( i \) is present in the transaction \( j \). Note that tid-lists are stored contiguously in increasing frequency order, i.e. the first row of the bitmap contains the least frequent item’s tid-lists. The same order is used during the depth-first visit of the lattice.

Once the vertical dataset is materialized, DCI-CLOSED uses a simple heuristic to understand the nature of the data. It is, in fact, an adaptive algorithm: depending on whether the data is dense or sparse, two different mining strategies are exploited. The heuristic has two stages [52][42]. If the average support of frequent singletons is larger than a fixed threshold then the data is considered dense. Otherwise, the algorithms calculates the identical portions of the bit-vectors corresponding to the most frequent items: if a sufficiently large subset of items has in common a sufficiently large portion of their bit-vectors, then the dataset is considered dense, and sparse otherwise.

As discussed before, when the dataset is sparse, the collection of frequent itemsets is very similar to the collection of closed ones. During the execution of a CFIM algorithm, large part of the closure computations and duplicate detection procedures will be useless and therefore a source of overhead. For these reasons, we decided to adopt a slightly modified \( k \)-DCI [42], which is an improved DCI algorithm that extracts efficiently frequent itemsets. The algorithm is level-wise, since exploiting the \textsc{Apriori} principle is very beneficial in sparse datasets. We introduced an additional \textit{closed-ness test}, that acts as a filter over the frequent itemsets discovered. Since a frequent \( k \)-itemset is closed if no superset with the same support exists, or equivalently, if it is not \textit{subsumed} by any \((k + 1)\)-itemset, we delay the output of the closed frequent \( k \)-itemsets (level \( k \)) until all the frequent \((k + 1)\)-itemsets (level \( k + 1 \)) have been discovered. Then, for each frequent \((k + 1)\)-itemset, we mark as \textit{non-closed} all subsumed \( k \)-itemsets. At the end of the \((k + 1)\)-th iteration, we can finally identify as closed all the frequent \( k \)-itemsets that result to be \textit{unmarked}.

We are not going to give further detail about this “sparse strategy” of the algorithm, since the most of our contributions are settled in the “dense strategy”. Deeper insights are given in \( k \)-DCI paper. However, in the experimental section we will show the goodness of this choice. Also, an interesting study [68] shows that in a peculiar and hard to mine synthetic dataset, having high average transaction length and closure based equivalence classes of size one, DCI-CLOSED is largely the best performing algorithm. Hereafter, we will only discuss the strategy adopted by DCI-CLOSED to mine dense datasets.

### 4.4.1 Detecting order-preserving generators

Rather than implementing a duplicate detection technique directly based on the closure operator as \textsc{Close}t+, we would prefer to use efficiently the Extension Lemma. Given a generator \( Y = X \cup i \), we can distinguish in the set of items \( I \) two disjoint subsets, defined as follows:
Definition 4.3 (Pro-oder and Anti-order sets) Let \( \prec \) be an order relation among the set of literals in the dataset, and let \( Y = X \cup i \) be a generator. We denote with \( Y^+ \) the pro-order and with \( Y^- \) the anti-order set of items for \( Y \) defined as follows:

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

The set \( Y^+ \) corresponds to the set of items that are still present in the projected dataset \( D_Y \) as it is built by Algorithm 4. Viceversa, \( X^- \) corresponds to the set of items that were removed from \( D_X \). In this sense, the items \( X^+ \) can still be used to discover items \( i \in c(X) \), while \( X^- \) cannot.

We are going to abandon for a while the recursive projection technique, since this is not used by DCI-Closed. However, we still want to detect a generator may lead to a duplicate, i.e. whose closure involve a backward jump in the lattice. We name the generator whose closure can be computed correctly only with the items in their pro-order set order-preserving generator.

Definition 4.4 (Order-preserving generator)
A generator \( Y = X \cup i \) is said to be order-preserving if and only if either \( Y = c(Y) \) or \( j \in (c(Y) \setminus Y) \Rightarrow j \in Y^+ \). Viceversa generator is not order-preserving if \( Y = c(Y) \) and \( j \in (c(Y) \setminus Y) \Rightarrow j \in Y^- \).

The exploitation of order-preserving generators can be very efficient when the data is stored in vertical format, i.e. when every tid-list \( g(i) \) is directly accessible, as in DCI-Closed. It is sufficient to compare \( g(Y) \) with \( g(i) \) for every \( i \in Y^- \), and as soon as an item that satisfies \( g(Y) \subseteq g(i) \) is discovered, the generator \( Y \) is known to be non order-preserving. Note that, even if Charm has a vertical representation of the dataset, it cannot exploit this lemma because of the pruning strategy described so far, that removes from projected datasets those tid-lists that we need.

According to our definitions, detecting non order-preserving generators by using the Extension Lemma is equivalent to detecting duplicate generator by exploiting the Subsumption Lemma. All the depth-first algorithms we described till now would traverse the frequent itemsets lattice in the very same way. The only differences reside in their own solutions to the duplicate detection problem.

Similarly to Closet+, our new approach brings several advantages. First, the correctness of the algorithm does not depend on the order according to which closed itemsets are discovered. The so called first generation algorithm, bases the detection of spurious itemsets on the fact that the corresponding closed itemsets were already discovered and stored in a global and evolving data structure. This implies that there is a strict order by which the lattice has to be visited. Conversely, this new (second generation) approach allows to explore the lattice in any order, since the only shared data structure is the dataset itself, which does not change over time. For instance, this approach allows a parallel implementation, since the depth-first
visit naturally identifies sub-problems that can be mined in parallel. We will come back to this in Chapter 6.

Conversely to Closet+, our algorithm makes this duplicate detection technique efficient. Closet+ needs to parse all the transactions supporting a given key-pattern in order to understand that this is not order-preserving, and this implies a number of jumps through the nodes in the FP-tree. The process is definitively too costly, and it is in fact used only with sparse datasets. The technique adopted by DCI-Closed is indeed novel, and it does not need to apply the closure operator directly, that means to intersect a large set of transactions. It uses the same Extension Lemma that is used to calculate the closure of an itemset via tid-list inclusion operations. These can be implemented with efficient bit-wise operations that provide good performance and high locality in memory access patterns. We found this technique to be very efficient especially in dense dataset where is more significant the problem of extracting closed frequent itemsets. In later sections we will see how it is possible to prune the pro-order and the anti-order sets and to optimize tid-list operations.

4.4.2 Computing closures

For what regards closures, these are also computed by applying the Extension Lemma, that is finding all those itemsets \( i \in Y^+ \) such that \( g(Y) \subseteq g(i) \). Note that even if the CHARM algorithm uses a vertical representation as DCI-Closed, our method is slightly different. CHARM uses linked-lists to store transactions, which implies that intersecting two tid-lists or checking whether one is included in another has the same cost. However, intersecting \( Y \)'s tid-list with the one of an item \( i \in Y^+ \) is more convenient, since, if \( i \notin c(Y) \), then the tid-list of \( Y \cup i \) is needed in the next step to generate a new candidate generator. For these reasons CHARM only performs tid-list intersections and never tid-list inclusions. Actually, it builds the whole projected dataset \( D_Y \).

On the other hand, since DCI-Closed actually exploits inclusions. It does not materialize the projected dataset \( D_Y \) to compute closures, thus saving memory. In addition, differently from other algorithms, it does not compute the support of \( Y \cup i \) if \( i \in c(Y) \), and this is important since counting the number of bits set in a bit-vector is expensive compared to the simple inclusion operation.

4.4.3 Mining dense datasets.

The pseudo-code of DCI-Closed is shown in Algorithm 5. As previously discussed, the dataset \( D \) is firstly scanned to determine the frequent items \( L_1 \), and to build the bit-wise vertical dataset \( BM \) containing the various tid-lists \( g(i), \forall i \in L_1 \).

The recursive procedure that visits the lattice of order-preserving generators has three main parameters: an order-preserving generator \( X \) and the corresponding pro-order \( X^+ \) and anti-order \( X^- \) sets. This is called for the first time with the arguments
4.4. The DCI-Closed algorithm

Given a valid generator, i.e. frequent and order preserving, the algorithm computes its closure (line 11). The resulting itemset \( c(X) \) is closed and frequent and it thus belong to the solution set \( C \). Note that it could be immediately output, since it is not needed for the successive iterations.

Then, the procedure builds all the possible generators \( Y \), by extending a closed itemset \( c(X) \) with the various items in the pro-order set \( i \in X^+ \) (line 17). Note that, even if in the items are chosen from \( X^+ \) according to the order \( \prec \) in the pseudo-code (line 16), this is not a general requirement since items may be chosen in whatever order, and this may be different for in each recursive invocation.

Given a candidate generator \( Y = X \cup i \) this must pass two tests to be considered valid. First it has to be frequent (line 19) in order to generate a closed frequent itemset. If it is not frequent it can be discarded as well as its supersets. Second, it
has to be order-preserving (line 20), in order not to generate a duplicates. Similarly, non order-preserving generators and their supersets can be discarded.

The new generator \( Y \), accompanied with the corresponding anti-order and pro-order sets \( Y^- \) and \( Y^+ \) is used to invoke a new recursion of the algorithm (line 21).

While the composition of \( Y^+ \) guarantees that the various order-preserving generators will be produced, the composition of \( Y^- \) guarantees that non order-preserving ones will be correctly detected.

Growing and pruning the pro-order set. Regarding the construction of \( Y^+ \), this must ensure the visit of the whole lattice of order-preserving generators, i.e. the completeness of the algorithm. In fact, a generator \( Y = X \cup i \) can be further extended all the items \( j \in X^+ \) but those \( j \preceq i \) in order to guarantee the order-preserving property. This is obtained in line 14 and considering that each item \( i \) used to build new generators is removed from the current \( Y^+ \) in line 16.

Some pruning is exploited at line 14, which removes the items belonging to the current closure, meaning that they have already taken in consideration and thus they would lead to duplicates in subsequent iterations. Finally, those items leading to infrequent or non order-preserving itemsets are discarded since the would not lead to any novel interesting closed itemset.

Growing and pruning the anti-order set. The composition of \( Y^- \) is symmetrical. Given a generator \( Y = X \cup i \), then all the items \( j \in X^- \) plus the ones \( j \in X^+ | j \prec i \) lead to non order-preserving generators. Also in this case, the presence of invalid generators introduces some pruning opportunities. Regarding an infrequent generator \( Y = X \cup i \), the item \( i \) can be evicted because it will not help in detecting non order-preserving generators \( Z \supset X \), since this would imply that \( g(Z) \subseteq g(i) \Rightarrow \sigma(Z \cup i) \geq \sigma \Rightarrow \sigma(X \cup i) \geq \sigma \) which is a contradiction.

If \( Y = X \cup i \) is frequent but non order-preserving, then there exists an item \( j \prec i | g(X \cup i) \subseteq g(j) \). We can show that \( i \) is redundant in next iterations, since for every \( Z \supset X \), if \( g(Z) \subseteq g(i) \), that is we can use \( i \) to prune \( Z \), then \( g(Z) \subseteq g(X \cup i) \Rightarrow g(Z) \subseteq g(j) \), which means that we can equivalently use \( j \).

Example 4.8 (Running example) We can show how DCI-Closed works by looking at the example of Figure 4.1. The visit of the lattice is illustrated in Figure 4.2. We comment every invocation of the recursive procedure Mine-Node.

\text{Mine-Node}(X = \emptyset, X^- = \emptyset, X^+ = \{abcd\}, \ldots). \text{The first order-preserving generator is trivially } X = \emptyset, \text{which is also closed since no item in } X^+ \text{ as support } D. \text{Four generators can be constructed by adding to } X \text{ the single items in } X^+ = \{a, b, c, d\}. \text{It easy to see that every new generator } Y = X \cup i \in X^+ \text{ is order preserving since the is not any item } j \prec i \text{ such that } g(i) \subseteq g(j). \text{Therefore the algorithm can continue the visit recursively.}

\text{Mine-Node}(X = a, X^- = \emptyset, X^+ = \{bcd\}, \ldots). \text{DCI-Closed checks whether } g(a) \text{ is set-included in } g(j), \forall j \in X^+(\text{i.e., } g(b), g(c), \text{and } g(d)), \text{and discovers that}
4.4. The DCI-Closed algorithm

Figure 4.2: The visit of the lattice performed by DCI-Closed our algorithm to extract all the closed frequent itemsets.

\[ c(Y) = c(a) = \{acd\}. \] The new closed itemset can be extended to produce a new generator \( Y = \{acd\} \cup \{b\} \). The anti-order set \( Y^- \) is equal to \( X^- \), and the items \{cd\} are removed from \( X^+ \) to create the pro-order set \( Y^+ \). \( Y \) is trivially order-preserving since \( Y^- \) is empty, and it is used to reiterate the algorithm.

\[ \text{Mine-node}(X = \{acd\}, X^- = \emptyset, X^+ = \{d\}, \ldots). \] The only remaining item in \( X^+ \) is \( d \), which belongs to \( c(X) \), resulting in a new closed itemset \{abcd\}. No item is left to continue the visit recursively.

Therefore, exploring the sub-tree rooted in \{a\}, DCI-Closed has carried out its first exploration of the search space, and has found two closed itemsets, first \{acd\} and then \{abcd\}. Note that these two closed itemsets have not been generated in lexicographical order, since and \{acd\} \neq \{abcd\}, but in extension order \{a\} \prec \{ab\}.

After some backtracking, the visit continues from \text{Mine-node}(X = b, X^- = a, X^+ = \{cd\}, \ldots). \] The closure of the order-preserving generator \( X \) is found by comparing \( g(b) \) with the tid-lists of the itemset in the pro-order set \( X^+ \). Since \( g(b) \subseteq g(d) \) we have that \( c(X) = \{bd\} \). This new closure is used to generate a new candidate \( Y = \{bd\} \cup \{c\} \). This time we can find that \{a\} \in \( Y^- = X^- \) is such that \( g(b) \subseteq g(a) \) and therefore \( Y \) is not order-preserving. The itemset \( Y \) is this removed, and DCI-Closed can backtrack since there is no remaining item in \( X^+ \).

\[ \text{Mine-node}(X = \{c\}, X^- = \{ab\}, X^+ = \{d\}, \ldots). \] The only items that can be
used to calculate the closure of \( X \) is \( \{d\} \), but \( g(c) \not\supseteq g(d) \) and therefore \( c(X) = X \). A new generator \( Y = c(X) \cup \{d\} \) is produced. Given \( Y^- = X^- \), the generator \( Y \) is not order-preserving since \( g(cd) \supseteq g(a) \). The itemset \( Y \) can be discarded.

**MINE-NODE**\((X = \{d\}, X^- = \{abc\}, X^+ = \emptyset, \ldots)\). Since there is no item in \( X^+ \), the current order-preserving generator is also closed. The new closure \( c(X) = \{d\} \) is the last closed itemset discovered.

### Optimizations saving bit-wise operations

DCI-CLOSED inherits the internal representation of our previous frequent set mining algorithms DCI and k-DCI. The vertical dataset is represented by a bitmap matrix \( BM_{M \times N} \) stored in main memory. The \( BM(i, j) \) bit is set to 1 if and only if the \( j \)-th transaction contains the \( i \)-th frequent single item. Row \( i \) of the matrix thus represents \( g(i) \), the tid-list associated with item \( i \).

Given a generator \( Y \), the basic operations performed by DCI-CLOSED are:

1. generating the tid-list \( g(Y) \)
2. calculating the support \( |g(Y)| \) of \( Y \);
3. performing duplicate check over \( Y \);
4. computing the closure \( c(Y) \).

To perform all these operations, DCI-CLOSED exploits efficient bit-wise operations. We exploit \( \text{AND} \) operations to compute \( g(Y) = g(X) \cap g(i) \), that is to intersect the tid-list of \( X \) with the one of \( i \), in order to derive the tid-list of \( Y \). The cardinality of \( |g(Y)| \) is computed by counting the number bits set in the bit-vector \( g(Y) \). For what regards the last two operations, we have to check the inclusion of \( g(Y) \) in the various tid-lists \( g(i) \) associated with single items \( i \) belonging either to \( Y^- \) or to \( Y^+ \). These inclusion checks are carried out by \( \text{AND} \)-ing word-by-word \( g(X) \) with \( g(i) \), and stopping at the first resulting word which is not equal to the corresponding word in \( g(X) \).

We adopted many optimizations to reduce the number of bit-wise operations actually performed by our algorithm. These optimizations, for the sake of simplicity, were not detailed in the pseudo-code of Algorithm 5.

The performance improvements resulting from each one of these optimizations are shown in Figures 4.4-4.3, which plots, for several datasets, the actual number of bit-wise \( \text{AND} \) operations performed DCI-CLOSED as a function of the support threshold. In particular, Figure 4.3 refers to the tid-list intersections to perform task (1), while Figure 4.4 deal with the inclusion checks to complete tasks (3) and (4). In all the plots, the top curves represent the baseline case, when no optimizations are exploited at all. From the top curves to the bottom ones, we incrementally introduced the optimization techniques detailed in the following.
First-level dataset projection. Let us consider the order-preserving generator \( X \) and its tid-list \( g(X) \). All the closed itemsets that are proper supersets of \( X \), will be supported by subsets of \( g(X) \). Thus, once \( X \) is mined, we can save work in the subsequent recursive calls of the procedure by projecting the vertical dataset \( BM \). This is carried out by deleting from \( BM \) all the columns corresponding to transactions not occurring in \( g(X) \). Since this bit-wise projection is quite expensive, we limited it to generators of the first level of recursion only, i.e. those \( Y = \emptyset \cup i \). In addition, we avoid projecting in correspondence of the small and the large \( i \): the former do not initiate a deep visit because they are the less frequent items, and the latter have a small number of children in the lexicographic spanning tree.

In the plots of Figures 4.3-4.4 we denoted this optimization with the label projection, and we compared its efficacy with the baseline case. We can see that the resulting number of bit-wise operations performed is about halved with respect to the baseline case.

Datasets with highly correlated items. The columns of \( BM \) are reordered to profit of data correlation, which entails high similarity between the rows of the matrix when dense datasets are mined. As in [42, 52], columns are reordered to maximize the size of a sub-matrix \( BM' \) of \( BM \) having all its rows and columns identical. Every operation (e.g. intersection) involving tid-lists within the sub-matrix \( BM' \) is performed by exploiting the equality of the its rows.

This is the most effective optimization regarding the intersection cost. In dense dataset sub-matrix \( BM' \) is likely to be large w.r.t. to tid-list size, and it includes the most frequent items. This means that many frequent itemsets are mined within this sub-matrix, and we can thus strongly reduce the actual number of intersection operation performed. We labeled this optimization with section eq in the plots of Figures 4.3-4.4. From the figure we can see that, exploiting this optimization, the number of operations performed further decreases of about one order of magnitude.

Reusing results of previous bit-wise intersections. Besides the above ones, we introduced another optimization which exploits the depth-first visit of the lattice of itemsets.

Let us consider the case in which we have find the items \( c(Y) \). The tid-list \( g(Y) \) must be compared with all the tid-lists \( g(j) \), for all items \( j \) contained in the \( Y^+ \). If \( Y \) results to be closed, then for all \( j \), we have that \( g(Y) \subsetneq g(j) \). However, we might discover that a large section of the bit-wise tid-list \( g(Y) \) is strictly included in \( g(j) \). Let \( g_h(Y) \) be the first \( h \) words of \( g(X) \), strictly included in the corresponding prefix \( g_h(j) \) of \( g(j) \). Hence, since \( g_h(Y) \subseteq g_h(j) \), it is straightforward to show that \( g_h(Z) \subseteq g_h(j) \) holds too, for every itemset \( Z \supset Y \). So, when we extend \( Y \) to obtain a new generator, we can limit the inclusion check of the various \( g(j) \) to the complementary portions of \( g_h(j) \). It is worth noting that, as soon as our visit of the itemset lattice gets deeper, the tid-list of an order-preserving generator \( X \) becomes
sparser, while the portions $g(X)$ strictly included in the corresponding portion of $g(j)$ gets larger, thus making possible to save a lot of work related to inclusion check.

In general, we can re-use results of partial inclusions also for what regards the detection of non order-preserving generator and also for the tid-list intersections.

In the plots of Figures 4.3-4.4 we denoted this optimization with the label included. As it can be seen from these plots, this optimization is very effective in reducing the cost of inclusion operations of about one order of magnitude.

Another interesting remark regards the comparison between the number of bit-wise intersection operations actually executed (Figure 4.3), and the number of operations carried out for the various inclusion checks (Figure 4.4). The two amounts, given a dataset and a support threshold, appear to be similar. This result might surprise the reader. One could think that the operations required for inclusion checks are the vast majority, since, for each frequent generator $Y$, we have to check the inclusion of $g(Y)$ with almost all the tid-lists $g(j)$. Indeed, we described a few pruning strategies that reduces the number of tid-lists $g(j)$ that need to be taken into consideration. Also, while tid-list $g(Y)$ must be computed for any generator $Y$, only those that survives the frequency pruning are further processed for inclusion checks. Moreover, whereas tid-lists must be completely scanned in order to produce $g(Y)$, the same does not hold for inclusion checks: we can stop the check as soon as a single word of $g(Y)$ results to be not included in the corresponding word of $g(j)$.

Also note that this “bit-wise” approach is very susceptible to improvements and optimizations. One example is the D-CLuB [41], an algorithm for mining frequent itemsets that uses a vertical bitmap representation of the dataset and and exploits interesting recurring patterns in bit-vectors.

### 4.5 Performance analysis

We tested DCI-Closed on a suite of publicly available datasets: Chess, Connect, Pumsb, Pumsb*, Retail and T40I0D100K. They are all real datasets, except for the last one, which is a synthetic dataset available from IBM Almaden. The first four datasets are dense and produce large numbers of frequent itemsets also for large support thresholds, while the last two are sparse. The characteristics of these datasets are reported in detail in Appendix A. The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory.

### 4.5.1 Performance comparisons

We compared the performances of DCI-Closed with those achieved by two well known state-of-the-art algorithms: FP-Close [28], and Closet+ [60]. FP-Close

---

64 4. DCI-Closed: a new algorithm for mining closed frequent itemsets
4.5. Performance analysis

Figure 4.3: Number of word intersections actually performed by the DCI-CLOSED when the various optimization techniques discussed in Section 14.3 are exploited. The plots refers to the Chess (a) and Connect (a) dataset. The number of operations is plotted as a function of the support threshold $\sigma$.
4. DCI-CLOSED: a new algorithm for mining closed frequent itemsets

Figure 4.4: Number of word inclusions actually performed by the DCI-CLOSED when the various optimization techniques discussed in Section 4.4.3 are exploited. The plots refers to the Chess (a) and Connect (b) dataset. The number of operations is plotted as a function of the support threshold $\sigma$. 
4.5. Performance analysis

Figure 4.5: Execution times (in seconds) required by FP-Close, Closet+, and DCI-Closed to mine various publicly available datasets as a function of the minimum support threshold.
4. **DCI-Closed**: a new algorithm for mining closed frequent itemsets

Figure 4.6: Execution times (in seconds) required by FP-Close, CLOSET+, and DCI-Closed to mine various publicly available datasets as a function of the minimum support threshold.
Figure 4.7: Execution times (in seconds) required by FP-Close, Closet+, and DCI-Closed to mine various publicly available datasets as a function of the minimum support threshold.
4. DCI-Closed: a new algorithm for mining closed frequent itemsets

Table 4.1: Relative percentages of time spent by DCI-Closed and FP-Close calculating supports, closures and in checking duplicates on various datasets.

<table>
<thead>
<tr>
<th>dataset (σ)</th>
<th>DCI-Closed</th>
<th>FP-Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess (25%)</td>
<td>26% 63% 11%</td>
<td>38% 60% 02%</td>
</tr>
<tr>
<td>Connect (6%)</td>
<td>32% 60% 08%</td>
<td>27% 70% 03%</td>
</tr>
<tr>
<td>Pumsb (45%)</td>
<td>41% 51% 08%</td>
<td>28% 70% 02%</td>
</tr>
<tr>
<td>Pumsb (8%)</td>
<td>56% 39% 05%</td>
<td>48% 47% 05%</td>
</tr>
</tbody>
</table>

is publicly available from the FIMI repository page [1] while the Windows binary executable of Closet+ was kindly provided us from the authors. We did not included Charm in our tests, because FP-Close was already proved to be faster [22]. The FP-Close and DCI-Closed algorithms were compiled with the gcc compiler available in the cygwin environment.

As shown in the plots reported in Figures 4.5-4.6-4.7, DCI-Closed outperforms both competitors in all the tests conducted. Closet+ performs quite well on the connect dataset with relatively high supports, but in all other cases it is about two orders of magnitude slower than DCI-Closed. FP-Close is effective in mining Pumsb*, where its performance is close to that of DCI-Closed, but it is one order of magnitude slower in all the other tests. It is worth noting that in order to fairly compare the different implementations, the above plots refer to tests conducted with support thresholds that allowed FP-Close and Closet+ never to run out of memory.

4.5.2 Time efficiency of duplicate check

In order to compare the efficiency of our duplication check technique with respect to that adopted by competitor algorithms, we instrumented the publicly available FP-Close code and our DCI-Closed code. Figure 4.8 shows the absolute times spent checking for duplicates while mining two dense datasets, Chess and Connect, as a function of the support threshold. As we can see, our technique remarkably outperforms the one used by FP-Close, with a speed-up factor close to six when small support thresholds are used.

In Table 4.1 we report the relative percentage of time spent by DCI-Closed and FP-Close performing each one of the three basic operations of the closed mining task: support count, duplication check, and closure computation. These percentages were computed by not considering the time spent by both implementations in other operations (e.g., input/output). In both algorithms, the time spent checking for duplicates is significant, but while in FP-Close the computation cost is, in some cases, dominated by the duplication checking task, the same does not hold

for DCI-Closed, in which the time is more evenly distributed between duplication check and support computations. If we also consider that DCI-Closed remarkably outperforms FP-Close in the total elapsed time, the advantage of using our duplicate checking technique is further highlighted.

The reasons of this good performance are discussed in depth in the following. The worst case of a duplicate check happens when a generator \( Y \) is order-preserving. In such case, every tid-list in \( Y^- \) will be scanned without finding any of them to be included in \( g(Y) \). Let \( l \) be the number of words of each bit-wise tid-list. Since \( Y^- \) is at most as big as \( L_1 \), then the worst case cost of this inclusion check is proportional to \( |L_1| \times l \).

However, as soon as a generator is recognized as non order-preserving, it is pruned without considering any more the other tid-lists associated with items in \( Y^- \). Moreover, the inclusion check can stop the scan of each tid-list at the first word for which the inclusion check fails. Finally, thanks to its multiple optimizations (see Section 4.4.3), DCI-Closed scans any tid-list only once along any path from the root to the leaf of the visit. This means that, even if, in principle, the cost of a single duplicate check operation involving a single tid-list in \( Y^- \) is directly proportional to \( l \), it turns out to be considerably lower due to these optimizations. In practice, every

![Figure 4.8: Absolute times (in seconds) spent checking for duplicates by DCI-Closed and FP-Close on datasets Chess and Connect, as a function of the support threshold.](image-url)
Figure 4.9: Amount of memory required by DCI-Closed, Closet+, and FP-Close for mining a dense (a), and a sparse (b) dataset, as a function of the minimum support threshold.
4.5. Performance analysis

Table 4.2: Per generator average number of bit-wise word intersections (PUMSB dataset).

| $\sigma$ | $|L_1|$ | $l$ | $|C|$ | ANDs |
|------|-------|-----|------|------|
| 40%  | 71    | 1533| 44,450,000 | 54.2 |
| 50%  | 51    | 1533| 7,120,000  | 38.5 |
| 60%  | 39    | 1533| 1,150,000  | 26.5 |
| 70%  | 34    | 1533| 240,000    | 23.8 |
| 80%  | 25    | 1533| 2,000      | 16.9 |
| 90%  | 20    | 1533| 1,400      | 16.5 |

duplicate check involves only a distinct and small portion of the tid-lists associated with items in the $Y^-\text{of each generator.}$

To support this claim, in Table 4.2, we reported the average number of operations, i.e., the bit-wise word intersections (ANDs), needed to perform the duplicate check for each encountered generator as a function of the minimum support threshold. Note that the number of operations per generator grows linearly with the size of $F_1$ as expected. However, the number of actual operations performed is considerably smaller than $|L_1| \times l$, since it resulted to be much less even than $l$, i.e., the size of a single tid-list (1533 words, in this case).

The table also shows, for each support threshold, the number of closed frequent itemsets extracted ($|C|$). Other algorithms hash the already mined closed itemsets to detect duplicates, but since $C$ grows exponentially, the hash performances may degrade suddenly as confirmed in Figure 4.8. Conversely, Tab. 4.2 shows that the number of operations performed by DCI-CLOSED is independent on the cardinality of $C$.

4.5.3 Space efficiency of duplicate check

Another issue related to the efficiency of a CFIM algorithm is the memory usage. Figure 4.9(a) plots memory requirements of FP-CLOSE, CLOSET+, and our algorithm DCI-CLOSED when mining the connect dataset as a function of the support threshold. Note that the amounts of main memory used by CLOSET+ and FP-CLOSE rapidly grow when the support threshold is decreased, due to the huge number of closed itemsets generated that must be stored in the main memory. Conversely, the amount of memory used by DCI-CLOSED, which does not need to maintain the frequent closed itemsets mined in memory, is nearly constant.

Let’s analyze the algorithm space requirements for the mining of dense datasets. We can measure the amount of memory used by the algorithm by taking into account all the information needed along the deepest computational path in the lattice from the root to the current node/itemset. First, the dataset is stored in a vertical bitmap $BM$ having size $O(|L_1| \times |D|)$. As discussed in section 4.4.3, this maybe projected at most once, resulting in an additional non larger bitmap of size $O(|L_1| \times |D|)$. Also,
4. **DCI-CLOSED: a new algorithm for mining closed frequent itemsets**

Figure 4.10: Execution times (in seconds) and memory requirements of DCI-CLOSED and FP-CLOSE for different support thresholds as a function of the size of the sample extracted from a large synthetic dataset.
the algorithm must store all the information needed at each node of the lattice, i.e.
the tid-list of corresponding itemset and the relative anti-order and post-order sets.
A tid-list has size $|\mathcal{D}|$, and since the path from the root to the current node is long
at most $|\mathcal{L}_1|$, we may need additional $O(|\mathcal{L}_1| \times |\mathcal{D}|)$ memory. The pro-order set
$Y^+$ has initially $|\mathcal{L}_1|$ elements at the root, and new pro-order sets of decreasing size
need to materialized at each node, thus resulting in a cost of $O(|\mathcal{L}_1| \times (|\mathcal{L}_1| + 1)/2)$.
Conversely, the anti-order set is initially empty, and increases at each recursive step
but, since items are added in a precise order to $Y^-$ we could use a vector just of size
$|\mathcal{L}_1|$ to store all the anti-order sets at the same time. Therefore the space complexity
of DCI-CLOSED algorithm is

$$O((3 \times |\mathcal{L}_1| \times |\mathcal{D}| + |\mathcal{L}_1| \times (|\mathcal{L}_1| + 1)/2 + |\mathcal{L}_1|)$$

since the number of transactions is usually much larger than the number of frequent
singletons, this cost is usually dominated by

$$O((3 \times |\mathcal{L}_1| \times |\mathcal{D}|)$$

Note that, in the worst case, the total number of closed itemsets should be $O(2^{|\mathcal{L}_1|})$, and therefore, for what regards the space complexity, it is a bad choice to
store the closed itemsets already mined in order to detect duplicates as in CLOSET
and CHARM. For these algorithms, the size of the output is a lower bound on
their space complexity. Conversely, the memory size required by DCI-CLOSED is
independent of the size of the number of closed itemsets extracted.

As soon as the quantities $|\mathcal{L}_1|$ and $|\mathcal{D}|$ are discovered after the first scan, we can
already fix a memory limit.

Note that DCI-CLOSED uses, however, a different method for mining sparse
datasets, based on a level-wise visit of the lattice and the same in-core vertical bit-
wise dataset. Independently of the technique adopted, mining sparse datasets should
not be a big issue from the point of view of memory occupation, because the number
and length of frequent itemsets do not explode even for very low support thresholds.
In Figure 4.9(b) we plotted the results of the tests conducted on $T_{40I10D100K}$, a
sparse dataset where CLOSET+ is supposed to use its upward checking technique.
Also in mining this sparse dataset, the memory requirements of DCI-CLOSED are
remarkably lower than those of its competitors. In this case, however, we can see that
DCI-CLOSED memory requirements are not constant due to the need of maintaining
in the main memory frequent $k$ and $(k + 1)$-itemsets.

Considerations on memory management techniques

State-of-the-art CFIM algorithms use a custom memory manager. For example,
consider FP-Close, at each node of the visit it builds a conditional dataset and
a conditional collection of closed itemsets, both of them containing a large number
of nodes that in principle need to be allocated and de-allocated one by one. This
would result in a huge number of system calls that affect significantly the running time of the algorithm.

FP-Close, and also Closet+ and CHARM, use a custom memory manager that allows to allocate large chunks of memory at once and then use portions of it as needed. More than that, it allows to free all the data structures that accompany a single node of the lattice with one call only.

DCI-Closed, does not need a custom memory manager mainly because of the staticity of its data structure. We analyzed the memory requirements of DCI-Closed, and we have seen it is possible to upper bound this requirements. As soon as the quantities $|L_1|$ and $|D|$ are discovered after the first scan and before the recursive mining starts, we can fix such memory limit. It is thus sufficient to pre-allocate a large chunk of memory, use it and re-use it during the computation without any need for complex memory management strategies, and their related overheads.

### 4.5.4 Scalability

In this section we discuss results of tests conducted in order to verify the algorithm scalability, which is strongly related to the amount of main memory exploited by an algorithm. In other words, the demand of main memory is the main factor that may limit scalability of in-core closed itemset mining algorithms.

In order to verify the scalability of DCI-Closed, we ran the algorithm on random samples of different sizes, taken from a large dense dataset created with the IBM generator. The synthetic dataset occupies 405 MB, and contains 300,000 transactions with an average length of 300. The number of distinct items is 1000, while the correlation factor 0.9.

The results of the tests conducted are reported in Figure 4.10, which shows execution times (Figure 4.10(a)) and memory requirements (Figure 4.10(b)) of DCI-Closed and FP-Close for different support thresholds, as a function of the size of the sample extracted from the synthetic dataset. Note that FP-Close failed in most cases, since it ran rapidly out of memory when the support threshold was lowered. FP-Close mined successfully only 1/10 of the dataset with a minimum support threshold of 50%, and was not able to mine even 1/100 of the dataset with a minimum support of 45%. On the other hand, DCI-Closed completed successfully in all the tests. We can see from the plots that both execution times and memory requirements grow linearly with the size of the dataset. This means that DCI-Closed can mine any dataset as long as its vertical representation fits in the main memory, disregarding the number of closed itemsets extracted.

This is also confirmed by the results reported in Table 4.3, which shows the number of closed itemsets discovered, and the execution times required by DCI-Closed to mine some dense datasets with an absolute support threshold of a single transaction. The tests with datasets Chess, Connect and Pumsb* completed correctly after a huge, but still reasonable for the first two, amount of time, also using the lowest
Table 4.3: Execution times required by DCI-CLOSED, and number of closed itemsets extracted from some dense datasets mined with the lowest possible support threshold.

<table>
<thead>
<tr>
<th>dataset</th>
<th>absolute $\sigma$</th>
<th>time</th>
<th>number of closed itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>1 transaction</td>
<td>2,791 sec.s</td>
<td>930,851,336</td>
</tr>
<tr>
<td>Connect</td>
<td>1 transaction</td>
<td>9,486 sec.s</td>
<td>1,414,804,579</td>
</tr>
<tr>
<td>Pumsb*</td>
<td>1 transaction</td>
<td>45 days</td>
<td>8,998,642,796</td>
</tr>
</tbody>
</table>

possible support threshold. The huge number of closed itemsets mined unequivocally demonstrates the limited scalability, with respect to the support threshold, of any other algorithms which, differently from DCI-CLOSED, need to store in the main memory the full set of closed itemsets.

4.6 Conclusions

In this chapter we have investigated the problem of mining closed frequent itemsets from transactional datasets. We reviewed the state-of-the-art algorithms like CHARM, CLOSET, CLOSET+, and FP-CLOSE, and we showed that they are all similar to each other.

We also showed that it is possible to mine efficiently dense datasets without storing the set of itemsets mined so far, differently from all the other algorithms. We introduced the concept of order-preserving generators. By using order-preserving generators it is possible to remove non closed itemsets by applying the Extension Lemma rather than by applying the closure operator or the Sub-sumption Lemma. Thanks to this technique and to a vertical bitmap storing the dataset, our proposed algorithm DCI-CLOSED proved to be very efficient and to have very interesting properties.

In fact DCI-CLOSED has a very small memory footprint, thus allowing to mine large datasets, and it allows a decomposition of the mining problem into independent sub-tasks. We focus on these properties in the following chapters.

As a result of the efficient strategies and optimizations introduced, DCI-CLOSED outperforms other state-of-the-art algorithms and requires orders of magnitude less memory when dense datasets are mined with low support thresholds. The in depth experimental evaluation conducted, demonstrates the effectiveness of our optimizations, and shows that the performance improvement over competitor algorithms – up to one order of magnitude – becomes more and more significant as the support threshold decreases. Finally, we accurately assessed DCI-CLOSED scalability, and we showed that it allows dense datasets to be effectively mined also with the lowest possible support threshold.
4. **DCI-Closed**: a new algorithm for mining closed frequent itemsets
Since datasets happen to be very large in many real life applications, it is mandatory to perform the mining task by using an external memory algorithm. Unfortunately, only a few algorithms were proposed for mining frequent itemsets out-of-core, and none for mining closed ones. Indeed, the contribution of this chapter is exactly to design an out-of-core CFIM algorithm.

The algorithm splits the original dataset into small partitions, mines each partition in main memory, and then merges partial results. We show that in principle the merging process is not trivial. However, our theoretical analysis allows to reduce the problem of merging partial results to an external memory sorting problem.

To the best of our knowledge, this is the first algorithm of its kind.

5.1 Introduction

It is true that real world datasets are very large. They are so large that the traditional FIM or CFIM algorithms are not able to maintain their own data structures in main memory.

A few FIM ad-hoc algorithms have been designed in order to work out-of-core, such as PARTITION [60] and DISKMINE [27]. They exploit a divide et impera strategy by subdividing the original dataset into partitions that can be separately mined in the main memory. Such out-of-core techniques can be profitably utilized also in case of severe space constraints, e.g. because users have limited capabilities in resource utilization. Consider, for example, a multi-user server, in which single user programs are disallowed to allocate all the main memory available, to avoid swapping all the others.

The problem of mining frequent closed itemsets out-of-core is tougher. Yet, no study on the mining of closed frequent itemsets in secondary memory has been done. This is the challenge we address here.

The property of being closed is a global property of an itemset in the context of the whole collection of frequent itemsets, or, equivalently, in the context of the whole dataset. Therefore, partition-based algorithms fail because this property cannot be
evaluated locally on a subset of the dataset, or on the portion of closed itemsets discovered in such partition. Analogously to traditional CFIM algorithm based on projections, by separately mining a subset of the original dataset, it is possible to generate spurious itemsets. Their closure is not complete because of missing information in the partition they were found.

Nevertheless, we adopted a divide et impera approach in our novel algorithm. A further step is introduced to correctly merge the partial results obtained, by detecting and removing such spurious, i.e. non closed itemsets. Though this task is in principle not trivial, a smart theoretical analysis allowed us to reduce the problem of discarding spurious itemsets to the one of sorting strings in external memory.

Finally, we chose DCI-Closed for extracting closed itemsets in each partition. The reason is that it is the only algorithm for which we can predict in advance the amount of memory it needs. Therefore, on the bases of this estimate, it is possible to create a convenient partitioning of the dataset.

5.2 Partitioning strategies

Typical approaches of out-of-core algorithms consist of three steps:

1. subdivide the input data into smaller chunks that can be entirely processed in main memory;
2. independently process each portion of data in main memory;
3. merge the local results, i.e. the outputs coming from the processing of each chunk of data.

The overall effectiveness of three-phase algorithm of this kind strongly depends on the first step. A bad partitioning may involve a complex merging process. Also, it could create a large number of data subsets, which usually involves some overhead. This last issue is also related with the choice of the algorithm in charge of mining each partition of the dataset.

For what regards the FIM problem, there are two important strategies for partitioning the input data: PARTITION and DiskMine. They use an orthogonal approach. The former partitions the input dataset into disjoint subsets of transactions. The latter induces a subdivision of the dataset on the basis of a partitioning of the search space. In the following sections we analyze both algorithms. However, note that DiskMine is the more interesting of the two because the correspondent merging process is simply a concatenation of the various local results.

Unfortunately, extending these two strategies to the CFIM problem is not easy. This is because CFIM algorithms need to exploit a global knowledge at any time of the computation in order to decide about the closed-ness of a given itemset. This global knowledge is materialized in the historical collection of closed itemsets.
5.2. Partitioning strategies

5.2.1 Partitioning the input dataset

This is the approach adopted by Partition, a level-wise apriori-like FIM algorithm that reads the database at most twice to generate all frequent itemsets. Partition is based on two main ideas. The first one is to divide the dataset in disjoint partitions that can fit in main memory one at the time, and the second one is that every frequent itemset must be frequent in at least one of these partitions.

First, the dataset is partitioned horizontally, and local frequent itemsets are mined separately from each partition. If an itemset is frequent in every partition, it is also frequent in the original dataset, and its support is calculated by summing up the local frequencies in each partition. If an itemset is frequent in only a subset of the partitions, then we do not have complete information about its support. In this case a second scan is required to retrieve the support in the partitions where such itemsets is infrequent, i.e. it was not extracted. Note that after this second scan, such itemsets may result to be infrequent globally.

Partition exploits a proper partitioning of the dataset, since it splits the dataset into disjoint subsets of transactions which cover the whole dataset. Each subset can be mined separately, but false positives, i.e. itemsets that are locally frequent in some partition but result to be globally infrequent, may be created. Returning to the CFIM problem, if we adopt a similar partitioning strategy an even worse problem arises with the closed-ness property. In fact, an itemset which is not closed in a partition may be closed when considering the whole dataset. This means that not only we have to discriminate between false and true frequent patterns (local versus global frequent patterns), but also between false and true closed patterns, i.e. globally closed itemsets that result not to be closed in some of the dataset partitions.

In [44], we have shown that it is however possible to reconstruct the whole set of global closed frequent itemsets even if some of them is not present in any of the local results. Suppose we have two partitions $D_1$ and $D_2$, and the two collection, $C_1$ and $C_2$, of the closed itemsets extracted from them. We proved that, the global solution is made by all the closed itemsets mined locally, plus the result of the intersections between any couple of itemsets in the cartesian product $C_1 \times C_2$. This result can be easily generalized to the case of $p$ partitions by first merging the two collections $C_1$ and $C_2$, then merging this partial result with $C_3$ and so on.

The cost of the merging step is however very high. Assuming that a naïve algorithm for merging two sets of partial results takes $|C_i| \cdot |C_j|$ time, we would have an overall complexity of $\prod_{i=1}^{p} |C_i|$ for $p$ partitions. The merging phase thus becomes rapidly intractable as the number of partitions increases.
5.2.2 Partitioning the search space

**DiskMine** is an FP-Growth based FIM algorithm. FP-Growth stores the transactions in a trie-like data structure named *FP-tree*. The initial FP-tree is then recursively projected item by item, thus visiting the whole lattice of frequent itemsets. The idea behind **DiskMine** is that, even if the whole dataset may be large, every projection on single items is likely to be very small. Therefore instances of FP-Growth can be run on these projections entirely in main memory. Differently from the horizontal partitioning technique, the set of itemsets mined from each projection produce a proper partitioning of the global collection of frequent itemsets with complete support information. Therefore there is no need for a post processing phase for merging the results or a second scan for calculating missing supports, but it is enough to gather local results.

The projection-based partitioning strategy used by FP-Growth may be used within any FIM algorithm. This works as follows. Given a total order \( \prec \) among single items \( I \), all transactions containing the first item \( i_1 \) are inserted in the first projection \( D_{i_1} \), then all the transactions containing the second item \( i_2 \) in the second projection \( D_{i_2} \), but deleting every occurrence of \( i_1 \), and so on. Finally, it is possible to independently mine frequent itemsets starting with \( i_1 \) from \( D_{i_1} \), then itemsets starting with \( i_2 \) from \( D_{i_2} \), and so on. Note that the results sets generated from the various projections are disjoint by construction.

More formally, let \( D_i \) be a projection-based partition of \( D \) over the item \( i \in I \), defined as follows:

\[
D_i = \{ t' = t \setminus \{ j \in t \mid j \prec i \} \mid t \in D \land i \subseteq t \}
\]

The projection \( D_i \) is thus built only from those transaction \( t \) in the original dataset that contain \( i \) by removing all the items preceding \( i \) according to the total order \( \prec \). Note there is a slight difference from the kind projection described in Section 2.3 because the item \( i \) is still present in \( D_i \).

**DiskMine** merges many of such projections together in order to minimize the number of partitions and therefore the number of disk accesses. A possible way is to combine partitions of datasets which have been projected over contiguous items in the total order \( \prec \). We thus indicate with \( D_{[x,y]} \) the projected dataset obtained by merging all the projected datasets \( D_i \), \( \forall i \in [x,y) \). Formally, we have that:

\[
D_{[x,y)} = \{ t' = t \setminus \{ j \in t \mid j \prec x \} \mid t \in D \land \exists i \in t \mid x \preceq i \prec y \}
\]

Given the sorted set of single items \( I = \{ i_1, \ldots, i_{|I|} \} \), we can thus create \( P \) partitions \( D_{[p_0,p_1)}, D_{[p_1,p_2)}, \ldots, D_{[p_{P-1},p_P]} \) of the dataset where \( i_1 = p_0 \prec p_1 \prec \ldots \prec p_P = i_{|I|} \), such that each partition can be mined entirely in main memory. Note that during the mining phase, a FIM algorithm must extract only those (lexicographically ordered) itemsets starting with an item in \( [x,y) \).

The above strategy guarantees the possibility of independently mining each projection in order to get the whole set of frequent itemsets. Unfortunately this does
not hold when mining closed itemsets. This is because each partition does not encompass knowledge about the global collection of closed itemsets, and therefore it is not possible to locally understand whether an itemset is globally closed or not.

Example 5.1 For example, consider Figure 4.1, which shows a dataset $D$ and its frequent closed itemsets extracted with $\sigma = 1$. Consider now that from $D$, we can build two projected datasets $D_{(a,b)} \equiv D_a$ (see Figure 5.1), and $D_{(b,d)}$ (see Figure 5.2), where $D_{(b,d)}$ is the projected dataset obtained by merging $D_b$, $D_c$, and $D_d$. When we extract frequent closed itemsets from the two projections, the closed itemsets mined from $D_{(b,d)}$ are incorrect. We can see that itemsets \{bcd\} and \{cd\} are locally closed in $D_{(b,d)}$, but they are not globally closed in $D$ since they are sub-sumed (see Lemma 4.1) respectively by \{abcd\} and \{acd\}, which can be extracted from $D_{(a,b)}$.

It is clear that we cannot decide locally whether an itemset in a a given projection $D_{[x,y]}$ is globally closed or not, because there is no knowledge about the occurrences of items preceding $x$ in the order $\prec$.

Note that this is different from the horizontal partitioning approach, where we do not have information about other transactions outside the current projection. With this search space partitioning, we just miss information about the items pruned out from the current projection.

However, it holds that every closed frequent itemset is mined in some of the projections eventually. Given one closed frequent itemset $X \in \mathcal{C}$, there must exist a partition $D_{[x,y]}$ such that $x \leq X \leq y$. Since such partition contains by construction
all the items of $X$ and all its supporting transactions, $X$ will be returned as a closed itemset from $D_{[x,y]}$.

If we denote with $C$ the set of closed itemsets of $D$, and with $C_1, \ldots, C_P$ the closed itemsets extracted from $P$ partitions of the original dataset $D$, then the following holds:

$$C \subseteq (C_1 \cup \ldots \cup C_P) \cup \{\emptyset\}.$$ 

Note that, since during the mining of each partition $D_{[x,y]}$, the mining algorithm extracts, by definition, only those lexicographically ordered itemsets starting with an item in $[x,y)$, the empty set can not be extracted from any projected partition, and therefore must be considered separately.

At first glance, it seems easier to use such search space partitioning and to remove some non-closed itemset from the result set, rather than using the horizontal partitioning and derive the solution set with an additional mining. In the next section, we will show that this intuition is true, by reducing the problem of finding such spurious itemsets to the one of external memory sorting.

### 5.3 Spurious itemsets detection in search space partitioning approaches

We have seen that some locally closed itemset may be non-closed globally. We refer to these non-closed itemsets as *spurious*. Every spurious itemset $X$ is simply a frequent itemset such that $X \neq c(X)$, and therefore it is an additional representative of the equivalence class of $c(X)$.
5.3. Spurious itemsets detection in search space partitioning approaches

In order to detect such redundant itemsets, we could use the usual duplicate detection technique based on the Sub-sumption Lemma 4.1. Given a itemset $X$ which is closed in the partition $D_{[x,y)}$, we must check whether it is sub-sumed by some other itemset $Y$ mined in some other partitions.

Unfortunately, this technique is very expensive, both in time and space. In time, because it requires searching the possibly huge set of closed itemsets mined so far for the inclusion of each spurious itemset. In space, because in order to efficiently perform set-inclusion checks, all the closed itemsets have to be kept in the main memory. Unfortunately, when low minimum support threshold are used, it may happen to extract a huge number of closed itemsets, so that maintaining them in main memory for searching purposes may become unfeasible. Even using some efficient memory resident data structures, like the one proposed in [36] (inspired to inverted files), would not help.

In the following, we introduce Lemma 5.1, which suggests a different and innovative technique for detecting spurious itemsets that can be efficiently implemented in an external memory algorithm.

**Lemma 5.1** Let $C$ be the collection of closed itemsets in the input dataset $D$, and let $C_1, \ldots, C_P$ be the collections of closed itemsets mined from the $P$ partitions of the original dataset $D$, respectively $D_{[p_0,p_1)}, D_{[p_1,p_2)}, \ldots, D_{[p_{P-1},p_P]}$, where $I = \{i_1, \ldots, i_M\}$ are sorted ascendingly according to some order $\prec$ and $i_1 = p_0 \prec p_1 \prec \ldots \prec p_P = i_M$.

If $X \in C_i$ and $X$ is not globally closed in $D$, then there must exist an itemset $Y \in C_j \neq i$ with $Y \supset X$ and $\sigma(X) = \sigma(Y)$ such that $X$ is a suffix of $Y$.

**Proof.** If $X \in C_i$ is not closed in $D$, then there must exist an itemset $Y \in C$ such that $Y \supset X$ and $\sigma(X) = \sigma(Y)$. Since $C \subseteq \{C_1, \ldots, C_P\} \cup \{\emptyset\}$ and since $X$ is closed in $C_i$, then there exist $C_j \neq i$ such that $Y \in C_j$. Let us focus on the items in $\{Y \setminus X\}$. By construction of the various partitions, these items may only precede the items in $X$. Thus, since $\forall i \in \{Y \setminus X\}, i \prec j, \forall j \in X$, we have that $X$ is a suffix of $Y$.

The above adds an important information that we use to detect spurious itemsets. If an itemset $X$ discovered in some partition is not closed, not only it is sub-sumed by another itemset $Y$ extracted by some other partition, but also it holds that $X$ is a suffix of $Y$.

**Example 5.2** Consider $C_1$ the closed itemsets extracted from $D_{[a,b)}$ (see Figure 5.1), and $C_2$ the closed itemsets extracted from $D_{[b,d)}$ (see Figure 5.2). Given a non globally closed itemset $X \in C_2$, e.g. $X = \{bcd\}$, by applying Lemma 5.1 we know that there must exists an itemset $Y \in C_1$ such that $X$ is sub-sumed by $Y$, and $X$ is a suffix of $Y$. This itemset actually exists, and it is $Y = \{abcd\}$.

The above lemma suggests a very simple method to identify spurious closed itemsets extracted from distinct partitions. This method is not expensive and can be efficiently implemented by using an external memory algorithm.
First of all, it is worth noting that given any two itemsets \( X \in C_i \) and \( Y \in C_j \) such that \( X \) is sub-sumed by \( Y \), if we sort \( X \) and \( Y \) in the reverse order of \( \prec \), then \( X \) is a \textit{prefix} of \( Y \). Thus, let us consider the list \( \mathcal{L}_X \) made with the reversely sorted items of the itemset \( X \) preceded by its support value. We can easily show that if \( \mathcal{L}_X \) is a prefix of \( \mathcal{L}_Y \), then \( X \) is sub-sumed by \( Y \). This condition in fact ensures that both the sub-sumption conditions, \( Y \supseteq X \) and \( \text{supp}(X) = \text{supp}(Y) \), actually holds.

In order to detect spurious itemsets, we materialize such lists \( \mathcal{L}_X \) from the sets of all the locally mined itemsets, and then we sort all the lists in ascending lexicographical order. This sorting is done in external memory by using a multiway merge-sort algorithm. We read chunks of lists \( \mathcal{L}_X \) in a buffer of predefined size. When the buffer is full, we sort it in-core before dumping it to the disk. Finally, a multi-way merge algorithm is applied to get a single sorted collection. Detection and removal of spurious itemsets can be done easily during the multi-way merge step: if itemset \( X \) is spurious, then the itemset \( Y \) that sub-sumes \( X \) can only have an associated \( \mathcal{L}_Y \) that comes immediately after \( \mathcal{L}_X \).

**Example 5.3** Consider the sets \( C_1 \) and \( C_2 \) of the closed itemsets extracted from \( D_{[a,b]} \) (see Figure 5.1), and \( D_{[b,d]} \) (see Figure 5.2), respectively. From them we obtain the following lists \( \mathcal{L}_X \):

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>local closed itemset</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{acd}</td>
<td>( \mathcal{L}_{acd} = 2, d, c, a )</td>
</tr>
<tr>
<td>1</td>
<td>{abcd}</td>
<td>( \mathcal{L}_{abcd} = 1, d, c, b, a )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>local closed itemset</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>{c}</td>
<td>( \mathcal{L}_c = 3, c )</td>
</tr>
<tr>
<td>3</td>
<td>{d}</td>
<td>( \mathcal{L}_d = 3, d )</td>
</tr>
<tr>
<td>2</td>
<td>{bd}</td>
<td>( \mathcal{L}_{bd} = 2, d, b )</td>
</tr>
<tr>
<td>2</td>
<td>{cd}</td>
<td>( \mathcal{L}_{cd} = 2, d, c )</td>
</tr>
<tr>
<td>1</td>
<td>{bcd}</td>
<td>( \mathcal{L}_{bcd} = 1, d, c, b )</td>
</tr>
</tbody>
</table>

Once the lists \( \mathcal{L}_X \) associated with the various itemsets \( X \) are built and stored on disk, we can sort them by using an external memory algorithm. In our example, we eventually obtain:

<table>
<thead>
<tr>
<th>list</th>
<th>is prefix</th>
<th>globally closed itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, ( D, C, B )</td>
<td>YES!</td>
<td>{abcd}</td>
</tr>
<tr>
<td>1, ( D, C, B, A )</td>
<td>no</td>
<td>{bd}</td>
</tr>
<tr>
<td>2, ( D, B )</td>
<td>no</td>
<td>{acd}</td>
</tr>
<tr>
<td>2, ( D, C )</td>
<td>YES!</td>
<td>{c}</td>
</tr>
<tr>
<td>3, ( D, C, A )</td>
<td>no</td>
<td>{d}</td>
</tr>
<tr>
<td>3, ( D )</td>
<td>no</td>
<td>{c}</td>
</tr>
</tbody>
</table>
Since the two lists \( L_{bcd} \) and \( L_{cd} \) result to be prefixes of lists \( L_{abcd} \) and \( L_{acd} \), respectively, which occur in the next position after reordering, the two associated itemsets \( \{bcd\} \) and \( \{cd\} \) can be safely discarded being spurious itemsets.

As we mentioned before, the closure of the empty set must be considered separately. Since if \( c(\emptyset) \neq \emptyset \), then \( c(\emptyset) \) would be mined from some partition, we must only consider the case corresponding to \( c(\emptyset) = \emptyset \). Note that \( c(\emptyset) \neq \emptyset \) only if the most frequent itemset appears in all the transactions of \( D \), i.e. \( \forall i \in c(\emptyset), \text{supp}(i) = |D| \).

Therefore we must add the empty set to the collection of globally closed itemsets only when no itemset has support equal to \(|D|\).

### 5.4 The mining engine: DCI-CLOSED

In our context we need a CFIM algorithm to process a single partition that meets two important requirements: the amount of memory used must be as low as possible and, more importantly, it must be predictable. Meeting both these requirements is a prerequisite to the possibility of devising an effective partition strategy able to produce dataset partitions that can be mined respecting a given maximum memory constraint. To the best of our knowledge, the only CFIM algorithm respecting the above requirements is DCI-CLOSED [43, 45].

DCI-CLOSED exploits a divide-et-impera strategy and a bitwise vertical representation of the database. It has been proved to outperform other state-of-the-art algorithms on most dataset, and furthermore, due to its space efficiency, it completes successfully the mining tasks on large input datasets and with low support thresholds that cause all the other algorithms to fail.

Thanks to its particular duplicate detection strategy, it can run efficiently without storing in main memory the large collection of the closed itemsets mined so far. This also means that the memory footprint of the algorithm does not depend on the size of its output. In fact, roughly speaking, DCI-CLOSED only needs to store the original dataset and the tid-lists corresponding to the node along the current path of the depth-first visit of the lattice. In Section 4.5.3 we showed that the space complexity of DCI-CLOSED is \( O((3 \times |L_1| \times |D|)) \).

### 5.5 Creating Partitions

Given a dataset \( D \), a minimum support threshold \( \sigma \), and a maximum memory size \( \hat{M} \), we must create a small number of partitions such that they can be entirely mined in at most \( \hat{M} \) bytes, according to the definition of partition given before.

For the sake of simplicity, we have chosen to build them by merging dataset projections on contiguous single items, according to some ordering \( \prec \), even if this does not guarantee to have the smallest possible number of partitions. We define \( \prec \) as the increasing support ordering of frequent items, because many studies suggest...
that such order reduces size of the search space. Of course, we discard infrequent items.

We can reformulate the problem as follows: given \( D \) and the ordered set of its frequent items \( L_1 \equiv \{ a_1, \ldots, a_{|L_1|} \} \), we must find a possibly small set of items \( \{ p_1, \ldots, p_P \} \) in \( L_1 \), such that the partitions \( D_{[a_1, p_1]}, D_{[p_1, p_2]}, \ldots, D_{[p_P, a_{|L_1|}]} \) can be mined using less than \( \hat{M} \) bytes.

Of course, the amount of memory required for mining frequent closed itemsets depends on the specific algorithm exploited. We chose DC1-Closed for its low and predictable memory requirements.

Let \( I_{[h,k]} \) be the set of frequent single items in the projection \( D_{[h,k]} \), then the memory required by DC1-Closed is bounded by \( (3 \cdot |I_{[h,k]}| \times |D_{[h,k]}|) \) bits. In order to reduce the size of a given projection, we perform some additional pruning. In fact, if we evaluate separately the closed-ness of the items in \( L_1 \), and we can use the projections to discover only frequent closed itemsets of length greater than 1. In particular, in every projection \( D_{[h,k]} \), we can ignore transactions of length 1, and discard those items \( x \in [h,k) \) such that no superset of \( x \) exists in \( L_2 \), where \( L_2 \) is the collection of frequent 2-itemsets in \( D \). Thus, we redefine both \( D_{[h,k]} \) and \( I_{[h,k]} \) as follows:

\[
D_{[h,k]} = \{ \ t' = t \ \backslash \ \{ j \in t \mid j < h \} \mid t \in D \land \exists i \in t|h \leq i < k \\
\land \ |t'| > 1 \}
\]

\[
I_{[h,k]} = \{ \ x \in [h,k) \ s.t. \ \exists X \in L_2 |X \supset x, \}
\]

In order to chose the best partitioning schema, we must estimate the cardinality of \( D_{[h,k]} \) and \( I_{[h,k]} \) for every potential partition \( D_{[h,k]} \), i.e. for each possible couple \( [h,k) \) of frequent items. These values can be easily calculated with a second scan of the dataset. Moreover, to compute \( D_{[h,k]} \) we need \( \binom{|L_1|}{2} \) counters, one for each possible couple \( [h,k) \), plus \( |L_1| \) counters for each \( D_{[h,h]} \), and to compute \( I_{[h,k]} \) we need further \( \binom{|L_1|}{2} \) counters, for a total of \( |L_1|^2 \). After each scan a partitioning decision is taken on the basis of the available data.

Note that the above technique requires \( |L_1|^2 \) counters to be stored in main memory. If \( |L_1| \) is large, it may happen that the required memory breaks the maximum limit \( \hat{M} \). If this is the case, in order to fulfill memory constraints, we perform multiple scans of the dataset, and during each scan consider only a subset of the \( |L_1|^2 \) counters.

Given all the possible values of \( D_{[h,k]} \) and \( I_{[h,k]} \), it is trivial to decide the partitioning. The algorithm starts considering \( D_0 \) and tries to add the next item \( i_1 \). If \( D_{[i_0,i_1]} \) does not fit \( \hat{M} \) then the first partition is \( D_0 \) and the algorithm re-iterates starting from \( D_1 \). Otherwise it is possible to check whether \( D_{[i_0,i_2]} \) may fit the memory limit. In case some projection on a single item \( D_i \) does not fit in main memory, it is possible to run OOC-Closed recursively on \( D_i \) itself.
5.6 OOC-CLOSED: an new algorithm for mining closed frequent itemsets from secondary memory

The pseudo-code of OOC-CLOSED, our out-of-core FCIM algorithm, is illustrated in Algorithm 6.

The first step relies on two scans of the dataset, during which the horizontal dataset is pruned, and decisions are taken concerning the number and size of partitions.

In the second step the various partitions $\mathcal{D}_{[k,k+n]}$ are mined by using DCI-CLOSED as CFIM mining engine. Note that DCI-CLOSED must extract from each partition $\mathcal{D}_{[k,k+n]}$ only the frequent closed itemsets whose first item belong to $[k, k + n)$. Moreover the list $\mathcal{L}_X$ associated with each closed itemset mined are written to disk for the following step.

Finally, the third step deals with removing non closed itemsets in order to obtain the exact result. It is carried out by means of the external memory sorting algorithm depicted in Section 5.3.

5.7 Experimental evaluation

We conducted a bunch of experiments on a Linux PC equipped with a 2GHz Pentium Xeon processor and 1GB of random-access memory. Three large datasets were used: WebDocs, USCensus1990, Synth2GB. Description of these datasets is given in Appendix A.

In Figure 5.3 we compare on the WebDocs dataset the performance of OOC-CLOSED with FP-CLOSE, and our in-core CFIM algorithm DCI-CLOSED. Note that we imposed OOC-CLOSED to run by using at most 30MB of memory. The aim of this test is to quantify the overhead introduced by our three-steps mining approach: partitioning, separate mining, and merging. From the plot reported in Figure 5.3(a) we can see that the execution times of the three algorithms are comparable for most of the support thresholds experimented. This means that the overhead introduced does not affect the overall performance remarkably, thus making our out-of-core approach not only viable in the cases where severe memory constraints really exist, but also efficient. Note that in the test conducted with the lowest support threshold, FP-CLOSE resulted very slow due to disk swapping activity. On the other hand, DCI-CLOSED always ran by using remarkably less memory than FP-CLOSE, thus justifying its choice as mining engine.

In Figure 5.4, we plotted the execution times, the number of partitions, and the amount of memory actually used by OOC-CLOSED for mining dataset WebDocs as a function of the memory threshold imposed. The performances of the algorithm resulted always to be very stable, since executions times did not increase signifi-
Figure 5.3: Running time and memory footprint of OOC-Closed, DCI-Closed and FP-Close on the WebDocs dataset.
5.7. Experimental evaluation

Figure 5.4: (a) Running time and memory footprint of OOC-CLOSED. (b) Number of partitions created and actual memory footprint of OOC-CLOSED. Both experiments were run on the WebDocs dataset and varying the maximum memory threshold $\hat{M}$. 
Figure 5.5: Running time and memory footprint of OOC-CLOSED with the US-Census1990 and Synth2GB dataset on varying the maximum memory threshold $\bar{M}$. 
Algorithm 6 OOC-CLOSED pseudocode

**Step 1: Partitioning.** Scan $\mathcal{D}$ twice to make decisions about projected partitions.

1: Scan $\mathcal{D}$ for the first time to find out the set of frequent items $\mathcal{L}_1$ and their supports, where $|\mathcal{L}_1|$.
2: Scan $\mathcal{D}$ for the second time. During the scan: (a) prune transactions on-the-fly by removing infrequent items, and re-map frequent items into the interval $[0, |\mathcal{L}_1|]$; (b) compute $\mathcal{L}_2$ and collect the information about memory occupancy of all possible partitions.
3: Choose the most suitable partitioning schema by considering the given memory constraint $\tilde{M}$, and save such information for the following step.

**Step 2: Mining.** Run DCI-CLOSED to extract frequent closed itemsets from all the partitions.

1: For each partition $\mathcal{D}_{[k,k+n]}$, DCI-CLOSED scans $\mathcal{D}$, creates on the fly an in-core (bitwise) vertical representation of $\mathcal{D}_{[k,k+n]}$, and mines from it all the closed itemsets whose first item belong to $[k, k + n)$. All closed itemsets mined are written to disk as lists $\mathcal{L}_X$.

**Step 3: Merging.** Remove spurious itemsets, and returns the final set of closed itemsets.

1: Run the external memory sorting algorithm to lexicographically order all the lists $\mathcal{L}_X$ stored on disk.
2: Remove non closed itemsets by discarding every list $\mathcal{L}_X$ that is a prefix of the list that occurs immediately after, and output the final result.

...cantly with the number of partitions. On the other hand, given the partitioning technique adopted, the number of partitions grows more than linearly as expected. More importantly, the plots show that the amount of memory actually used during execution always resulted lower than the memory threshold imposed.

Finally, Figure 5.5 reports the results of the tests conducted on datasets US-Census1990 and Synth2GB. Also in these tests the memory threshold was always respected by OOC-CLOSED.

5.8 Conclusions

We have presented a novel algorithm able to mine all the frequent closed itemsets from a transactional database using a limited amount of main memory. To our best knowledge, this is the first external memory algorithm for mining closed itemsets.

The two main contributions of this paper are, on the one hand, the optimization...
of an already known projected-based partitioning technique adapted to our framework, and, on the other hand, an innovative merging technique of the local results extracted from each partition.

We have shown how the exploitation of such partitioning technique requires a double scan of the dataset to collect enough information to decide how to subdivide it in order to obtain projected partitions that fit the available memory. Such information is also used to prune further the dataset and its partitions.

The main issue we have had to solve regards the possible generation of spurious frequent itemsets, which can be obtained if we simply combine the local results obtained from the separate mining of the partitions. We may in fact generate some additional frequent itemsets besides the truly closed ones. This unpleasant behavior is due to the partial knowledge available in each projected partition. This does not permit us to check, during the local mining of a partition, whether a produced itemset is globally closed or not. We have solved this problem in an elegant way. We have devised a novel out-of-core technique, based on a new theoretical insight, for merging the various local results and removing spurious itemsets. In particular, we have reduced the problem of merging partial solutions to an external memory sorting problem.
6

Harnessing modern computer architectures with \textbf{MT-Closed}: the first parallel CFIM algorithm

Inspired by emerging multi-core computer architectures, in this chapter we present \textbf{MT-Closed}, a multi-threaded algorithm for closed frequent itemset mining (CFIM). To the best of our knowledge, this is the first parallel CFIM parallel algorithm proposed so far.

We analyze which are the issues in parallelizing state-of-the-art CFIM algorithms, that is the possibility to exploit a subdivision of the search space into independent sub-problems. We argue that \textbf{DCI-Closed} is probably the algorithm that may best fit a parallel framework. In fact, \textbf{MT-Closed} is based on \textbf{DCI-Closed}.

Finally, we show how \textbf{MT-Closed} efficiently harnesses modern CPUs: it takes advantage of the presence of multiple cores, it exploits SIMD extended instruction sets, and it has a cache-friendly memory access pattern.

6.1 Introduction

We can recognize some important trends in the design of future computer architectures. The first one indicates that the number of processor cores integrated in a single chip will continue to increase. The second one is that memory hierarchies in chip multi-processors (CMPs) will become even more complex, and their organization will try to maximize the number of requests that can be served on-chip, thus reducing the overall number of cache misses that involve the access to memory levels outside the chip. Finally, nowadays modern processors have SIMD extensions, like Intel’s SSE, AMD’s 3DNow!, and IBM’s AltiVec, and further instructions and enhancements are expected to be added in future releases. Thus algorithm implementations can take large performance advantages from their use.

In order to maximize the utilization of the computing resources provided by CMPs, and increase application performance, it is mandatory to use multiple threads within applications. Programs running on only one processor will not be able to take full advantage of such future processors, since they will only exploit a limited
amount of implicit instruction-level parallelism (ILP). As a result, explicit parallel programming has suddenly become relevant even for desktop and laptop systems. Non trivial programming paradigms must be adopted: coarse-grained parallelism may result in high pollution of private caches, while a fine-grained parallelism may cause a large number of conflicts in caches shared levels.

Unfortunately, as the gap between memories and processors speeds continues to widen, more and more cache efficiency becomes one of the most important ingredients of performance. High performance algorithms must be conscious of the need of exploiting locality in data layout and, consequently, in data access. The presence of multiple processors accessing complex cache hierarchies having some shared levels increases the chance of cache conflicts. The side-effects of multi-threaded programming on this platforms are still unclear: in principle, increasing the number of CPUs may significantly reduce the benefits given by cache memories.

Furthermore, it is still unexplored the opportunity to exploit the SIMD co-processors that equip most modern CPUs. We instead think that the SIMD paradigm can be useful to speed-up several data mining sub-tasks that operate on large amounts of data.

In this chapter we discuss how a demanding algorithm to solve the general problem of closed frequent itemsets mining can be effectively implemented on modern computer architectures, by capturing all the technological trends mentioned above. The same CFIM algorithms that we have seen in previous chapters, are here deeply analyzed with respect to their capability to harness modern architectures.

We introduce a new algorithm named MT-Closed. This algorithm is based on DCI-Closed, which nicely fits a parallel framework. We show that MT-Closed is able to sub-divide the CFIM problem into independent sub-tasks that can be easily run in parallel. Moreover, the data structures used by MT-Closed, and the way they are accessed, exhibit high locality, especially spatial one. Consider that, the most accessed data structure adopted by MT-Closed is read-only, and therefore, the concurrent use of such data by multiple processors does not need synchronizations and does not cause cache incoherence. Finally, we show that the computational kernel of the MT-Closed algorithm is mainly based on bit-wise operations performed on long bit-vectors. Such operations are data parallel, and can be efficiently programmed using SIMD co-processors of modern CPUs.

These peculiarities characterize MT-Closed, an efficient, shared memory parallel CFIM algorithm for SMPs or multi-core CPUs. This is the main contribution of our work, since MT-Closed is, to the best of our knowledge, the first parallel CFIM algorithm proposed so far.

We also explore diverse scheduling/assignment strategies typical of parallel algorithms. In order two provide a good load balance we use a dynamic decomposition of the search space together with a dynamic assignment of the correspondent mining sub-task. This resembles a receiver-initiated work-stealing policy, where each idle thread can choose a different running thread which will release half of its mining sub-task.
The experiments show that MT-CLOSED may obtain very good speed-ups on SMPs/multi-core architectures with up to 8 CPUs.

6.2 Parallel pattern mining on modern computer architectures

To the best of our knowledge, no parallel algorithm has been proposed so far to mine closed frequent itemsets. On the other hand, many sequential CFIM algorithms were proposed in the past, and also some interesting parallel implementation of FIM algorithms exist as well.

The lack of parallel a CFIM solution is probably due to the fact that mining closed itemsets is more challenging than mining other kind of patterns. As discussed in previous chapters, during the exploration of the search space, a duplicate detection process is involved, and deciding whether or not a pattern leads to a duplicate, needs a kind of global knowledge either of the dataset or of the collection of closed patterns mined so far. This makes tougher the decomposition of the mining task into independent, thus parallelizable, sub-problems.

In this section, we review again some existing CFIM algorithms, by dropping them in this new framework. We weigh pros and cons of the parallelization of these algorithms, taking into consideration the capability to take advantage of modern CPUs.

6.2.1 First generation algorithms

Within this category of algorithms we include CHARM, CLOSET and FP-CLOSE. These algorithms exploit the sub-sumption Lemma 4.1 in order to detect whether the closure of an itemset $Y$ was already computed. An itemset leads to a duplicate if it is a subset of an already mined closed itemset having identical support. Note that this strategy requires a global knowledge of the collection of closed frequent itemsets mined so far. Thus, a historical collection of itemsets is queried and updated continuously during the mining.

This approach has several drawbacks when we think to its parallelization. Basically, parallelization involves the partition of the lexicographical spanning tree of the frequent itemsets lattice, into a set of non overlapping subtrees. This are assigned to a set of threads performing the mining task. According to this paradigm, an implementation based on the above algorithm would incur in the following problems.

Spurious itemsets. First generation algorithms use the historical collection of closed itemsets to detect duplicates. This requires a precise order in the visit of the search space. But, When different portions of the search space are mined in parallel, closed itemsets are discovered in an unpredictable order.
Given a closure based equivalence class, a non order-preserving key-pattern, whose projected dataset does not contain all the information needed to correctly compute its closure, may be discovered before the order-preserving key-pattern of the same class. Therefore, the Sub-Sumption Lemma is not exploited correctly since the correspondent closed itemset was not yet extracted. A spurious closed itemset is eventually computed and inserted in the historical collection of closed frequent itemsets.

Example 6.1 Consider the two key-patterns \( Y = \{a\} \) and \( Z = \{cd\} \). If these are mined according to the lexicographical order, then \( Y \) is discovered first, its closure \( c(Y) = \{acd\} \) is correctly computed in \( D_a \) and stored. When \( Z \) is generated, this is found to be subsumed by \( c(Y) \) and discarded.

Conversely, if we can make no assumption on the visiting order, then \( Z \) may be visited first. In this case, it would not be subsumed by any other itemset, and the closure computation in \( D_{cd} \) would produce the spurious itemset \( \{cd\} \neq c(Z) = \{acd\} \), since no information about the item \( a \) is present in the projected database.

Search space overhead. When a non order-preserving key-pattern is not detected, this is used to reiterate the mining, thus producing other spurious itemsets. Each sub-problem will be likely to explore a portion of the search space that is larger than needed, since it possibly contains lots of non order-preserving key-patterns and spurious itemsets. Therefore, the cumulative cost of all the mining sub-tasks becomes greater than the serial cost of a CFIM algorithm.

Maintenance. In principle, the historical collection of closed itemsets should be shared among every task. Each of them will search for supersets and insert new itemsets. Since insertion requires exclusive access, this may become a bottleneck because of the limited parallelism.

Also consider that the spurious itemsets generated must be eventually removed. A proper procedure has to be in charge of detecting and removing all the non closed itemsets. For what regards FP-CLOSE, this problem is replicated for each projected collection of closed frequent itemsets.

6.2.2 Second generation algorithms

In this category we include those algorithms that do not need to maintain the historical collection of closed itemsets already mined in order to detect non order-preserving key-patterns. These are \( \text{CLOSET}^+ \) and \( \text{DCI-CLOSED} \). They solve previous algorithms issues regarding their large memory footprint. The whole collection of closed frequent itemsets in fact occupies a non trivial amount of memory. Consider, for example, that the fast algorithm FP-CLOSE stores as many projections of such collection as number of the levels in the visit through lexicographic tree. Thus, it is
not rare for FP-CLOSE to run out of memory, and start swapping projected fp-trees in and out from secondary memory, with unacceptable performance degradation. The same consideration holds for what regards the cache usage: projected datasets would be continuously moved in and out the cache memory.

CLOSET+ introduces the upward checking technique, which applies the closure operator \( c(\cdot) \) directly on the dataset, thus being able to detect spurious itemsets. Also DCI-CLOSED uses the dataset itself, exploiting the Extension Lemma. While the former algorithm uses an horizontal representation of the dataset and consequently computes \( c(\cdot) \) by intersecting transactions, the latter algorithm uses a vertical representation of the dataset and exploits inclusion checks against some items’ tid-lists. Note that we are referring to the strategy adopted by DCI-CLOSED for dense datasets. In the remainder we will continue to refer to such “dense” strategy whenever referring to DCI-CLOSED.

These kind of algorithms require a global knowledge of the dataset, without any other additional dynamic data structure to be maintained. Such a global representation of the dataset is static, and therefore can be easily shared by many working threads mining different regions of the search space at the same time. Also, since accesses to this static data are read-only, run conditions never happen and threads may execute independently without any synchronization overhead. Finally, since the removal of spurious itemset is based on the dataset, which is always up-to-date, spurious itemsets are never generated with no search space overhead.

For these reasons, both DCI-CLOSED and CLOSET+ are suitable for the design of a parallel CFIM algorithm. In fact, the upward checking approach was used successfully in other parallel mining algorithms, such as PAR-CSP [17] for the parallel mining of closed sequential patterns.

However, the CLOSET+ algorithm is orders of magnitude slower than DCI-CLOSED. Also, the upward checking technique was actually used by the authors only with sparse datasets. This is why we chose to focus on DCI-CLOSED, which has the same advantages of CLOSET+, but with largely better performances for what regards the extraction of closed frequent patterns as shown in Chapter 4.

6.2.3 Further considerations on CFIM algorithms

To further support our choice of DCI-CLOSED, we address some other important issues. In fact, to fully benefit of modern computer architectures, we must consider carefully also the memory access patterns and SIMDization capabilities of these algorithms. Also from this point of view, fp-tree based algorithms do not seem appealing.

Pattern mining algorithms are memory intensive: they perform a large number of memory operations (about 60%) due to the several traversals of the dataset representation. Cache misses can thus considerably degrade the performance of these algorithms. In particular, FP-GROWTH-like algorithms are subject to the pointer-chasing phenomenon: the next datum is only available through a pointer.
6. Harnessing modern computer architectures with MT-Closed

The CLOSET+ approach obviously augments this problem. The authors of [21] improved the FP-GROWTH algorithm introducing a tiling technique. The basic idea is to re-organize the paths of the FP-tree in tiles, i.e. consecutive memory locations small enough to fit into the cache, and to maximize their usage once they have been loaded. Unfortunately, in the case of closed itemsets, the tiling benefit would in fact be wasted by omnipresent duplicate checks, that require a non localized access to a large data structure, whatever of the two lemmas is exploited. Conversely, the vertical bitmap representation of DCI-CLOSED provides high spatial locality of access patterns, and this cache-friendliness carries a significant performance gain.

Also, having multiple threads running on the same machine, multiplies the memory footprint of the algorithm. While DCI-CLOSED is very prominent, needing only a small amount of memory, first generation algorithms such as FP-CLOSE have stronger requirements due to the need of handling multiple dataset projections at the same time. Thus, the introduction of multiple threads will largely harm such algorithms, pushing them over the limit of the available memory.

Finally, DCI-CLOSED uses a bitmap representation of the dataset and bit-wise tid-lists intersection is the basic operation for calculating closure and supports. This means that SIMD operations working in parallel on string 128-bits long can easily be plugged in to the algorithm achieving yet another improvement. The same does not hold with FP-GROWTH-like algorithms, were the extended instruction sets, intended to improve floating point and integer operations, would not find a role.

In conclusion, DCI-CLOSED is faster than other CFIM algorithms, and it is the only that can be reshaped into a parallel algorithm, that not only takes advantage of multiple core in modern CPUs, but also of their extended instruction sets, still exhibiting a good behaviour in its memory requirements and access patterns.

6.3 The MT-CLOSED algorithm

This section describes the main aspects of MT-CLOSED, our parallel version of DCI-CLOSED.

DCI-CLOSED and MT-CLOSED are identical for what regards the first two iterations, were the most important data structures are initialized. In the first scan frequent single items, denoted with $L_1$, are extracted. Then, infrequent items are removed from the original dataset, and the new resulting set of transactions is stored in a vertical bitmap $B_M$. The row $i$ in the resulting bitmap is a bit-vector representation of the tid-list of the $i$-th least frequent item.

We chose not to take into consideration the parallelization of this first part of the algorithm. The main reason is that it would not impact on the overall performance of the algorithm for very expensive data mining tasks. Also, the most challenging issues dwell in the next part of the algorithm, were the real mining takes place.

In DCI-CLOSED, the recursive visit of the lattice of order-preserving key-patterns is implemented by the MINE-NODE procedure in Algorithm 5. This, as the recursion
progresses, grows a local data structure where the necessary per-iteration information is maintained, other than accessing to the shared vertical bitmap $B_M$. Such information includes the current node $X$, the sets of anti-order and pro-order sets $X^-$ and $X^+$, and the tid-list of the current node $g(X)$.

We refer to $J = \langle X, X^-, X^+, g(X), B_M \rangle$ as the job description, since it provides an exact snapshot of what the algorithm is doing and of what data it needs at a certain point of the execution. Moreover, due to the recursive structure of the algorithm, the job description is naturally associated with a mining sub-task corresponding to an invocation of the Mine-Node procedure.

It is possible to partition the whole mining task into independent regions, i.e. sub-trees, of the search space, each of them described by a distinct job descriptor $J_i$. In principle, we could split the entire search space into a set of disjoint re-
gions identified by $J_1, \ldots, J_m$, and use some policy to assign these jobs to a pool of parallel threads. Moreover, since the computation of each $J_i$ does not require any co-operation with other jobs, and does not depend on any data produced by other threads (e.g. the historical closed itemsets), each task can be executed in a completely independent manner. In the following we will see how to define and distribute $J_1, \ldots, J_m$.

In Figure 6.1, we give a pictorial view of our parallel algorithm. We can imagine a public shared memory region, where the vertical dataset is stored: the bitmap is accessed read-only and can be thus shared without synchronizations among all the threads. At the same time, each thread holds a private data structure, where new job descriptions are materialized for every new closed itemset encountered along the path of lattice visit.

We argument that this net separation between thread private and read-only shared data perfectly fits the multi-core architectures that inspired our work, and in particular their cache memory hierarchies. In the near future, not only it is expected that cache memories will increase their size, but we will also get used to quad- eight-core CPUs with complicated and deep cache memory hierarchies, where different levels of caches will be shared among different subsets of CPUs.

6.3.1 Load partitioning and scheduling

Parallelizing frequent pattern mining algorithms is a non trivial problem. The main difficulty resides in the identification of a set of jobs to be later assigned to a pool of threads providing a good load balancing.

We have already seen that, with DCI-CLOSED, it is easy to find a decomposition of the CFIM problem into independent tasks, however this may be not sufficient. One easy strategy would be to partition the frequent single items and assign the corresponding jobs to the pool of threads.

**Definition 6.1 (Naïve Lattice Partitioning based on literals)**

Let $\mathcal{L}_1 = \{i_1, \ldots, i_{|\mathcal{L}_1|}\}$ be the set of frequent items in the transactional dataset $\mathcal{D}$. The lattice of frequent itemsets can be partitioned in to $|\mathcal{L}_1|$ non overlapping sets: the itemsets starting with $i_1$, the itemsets starting with $i_2$, etc. Each is identified by a job description defined as follows: $J_{i_h} = (X = \{i_h\}, X^- = \{i \in \mathcal{L}_1 \mid i < i_h\}, X^+ = \{i \in \mathcal{L}_1 \mid i \not< i_h\}, g(i_h), BM)$, for each $i_h \in \mathcal{L}_1$.

Unfortunately, it is very likely that one among such jobs has a computational cost that is much higher than all the others. The difference may be such that its load is not comparable with the cumulative cost of all the other tasks.

Among the many approaches to solve this problem, an interesting one is [17]. The rationale behind the solution proposed by the authors, is that there are two ways to improve the naïve scheduling described above. One option is to estimate the mining time of every single job, in order to assign the heaviest jobs first, and
6.3. The MT-Closed algorithm

later the small ones to balance the load during the final stages of the algorithm. The other is to produce a larger number of tasks thus providing a finer partitioning of the search space: having more tasks provides more degrees of freedom for the job assignment strategy. Their solution merges these two objectives in a single strategy. First the cost of the jobs associated to the frequent singletons is estimated by running a mining algorithm on significant (but small) samples of the dataset. Then, the more expensive jobs are split on the bases of the 2-itemsets they contain.

Our choice is to avoid the expensive pre-processing step where dataset samples have to be determined the assignment to different threads.

We will materialize jobs on the basis of the 2-itemsets in the cartesian product $\mathcal{L}_1 \times \mathcal{L}_1$ as follows:

Definition 6.2 (Lattice Partitioning based on 2-itemsets)

Let $\mathcal{L}_1 = \{i_1, \ldots, i_{|\mathcal{L}_1|}\}$ be the set of frequent items in the transactional dataset $\mathcal{D}$. The lattice of frequent itemsets can be partitioned in to $(\binom{|\mathcal{L}_1|}{2})$ non overlapping sets: the itemsets starting with $i_1, i_2$, the itemsets starting with $i_1, i_3$, etc. Each is identified by a job description defined as follows: $J_{hk} = \langle X = \{i_h, i_k\}, X^- = \{i \in \mathcal{L}_1 \mid i < i_k\}, X^+ = \{i \in \mathcal{L}_1 \mid i \notin \{i_h, i_k\}\}, g(\{i_h, i_k\}), BM \rangle$, for each $i_h, i_k \in \mathcal{L}_1$.

Since the total number of tasks $(\binom{|\mathcal{L}_1|}{2})$ is typically much larger than the number CPU/cores of modern architectures, this partitioning strategy provides good chances to distribute evenly the load among the various processing units.

Note that not all the itemset in $\mathcal{L}_1 \times \mathcal{L}_1$ are frequent order-preserving generators. They will be immediately tested, and some of them will be eventually discarded as empty jobs. This is not a search space overhead due to the parallelism of the algorithm, since also the serial version would have to check all those itemsets and prune infrequent or non order-preserving ones.

For what regards the load partitioning and scheduling policies, we investigated two different approaches and all their possible combinations. In particular, we considered Static and Dynamic solutions for both the load partitioning and scheduling problems. We thus implemented three slightly different versions of MT-Closed.

Static Partitioning / Static Assignment (SS)

According to Definition 6.2 each job description $J_{hk}$ is used to instantiate a mining sub-task. Such statically fixed job descriptions are statically assigned to the pool of threads in a round-robin fashion. We will use this simple strategy as a sort of baseline; as discussed above, non static strategies are supposed to perform much better.

Static Partitioning / Dynamic Assignment (SD)

The job instances are the same as in the SS solution, but the assignment of each mining task is initiated by the working threads. Whenever a thread reaches the
idle state, it dynamically self-schedules the next available sub-task and materializes the necessary information, such as pro-order and anti-order sets, needed to start the mining. The SD strategy requires a limited number of synchronizations among the threads, since there is only a trivial critical section regarding the management of the couple of items \((i_h, i_k)\) identifying the next available job. Note that, since jobs are created in increasing order of the corresponding itemset \(\{i_h, i_k\}\), the last assigned jobs are also the least expensive, being associated with pro-order sets having a small cardinality. These cheap sub-tasks thus help in balancing the load during the final stages of the algorithm.

**Dynamic Partitioning / Dynamic Assignment (DD)**

This totally dynamic policy is based on a work stealing principle. Once a thread has completed its sub-task, it steals half of the remaining work of an heavy loaded thread. A thread is considered heavy loaded if at least \(l\) items are present in any of its pro-order sets. Recall that each recursion of the mining creates a new job description. Thread polling order is established locally on the basis of a simple round-robin policy so that the most recent victim of a theft is the less likely to be stolen again by the same thread.

One thread obtains a new mining job by copying from another thread a closed itemset \(X\), the corresponding tid-list \(g(X)\), the second half the set of items in \(X^+\) not yet explored by the robbed thread, and finally, an adjusted \(X^-\). At the same time, the pro-order set of the victim must be updated. This also means that private job-descriptors are not private anymore: these become a critical section that can be modified by any thread.

We designed this totally dynamic solution since experimental evidence showed that in some cases a very few sub-tasks results to be significantly more expensive than the others, and therefore they may harm the load balancing of the system. This DD solution has the great advantage that all the threads work until the whole mining process is finished, since there will be at least one other running thread from which to steal some workload. On the other hand, additional synchronization is required that may slightly slow down the algorithm.

### 6.3.2 First-level dataset projection

In Section 4.4.3 we explained that DCI-CLOSED adopted a particular choice for what regard dataset projections. Projections reduce the amount of information to be dealt during each iteration. However, the cost of projecting becomes significant when applied at every node of the visit. DCI-CLOSED exploits projections only for a subset of the order preserving key-patterns of length 1. This heuristic provides a substantial speed-up.

MT-CLOSED must adopt some projection strategy as well. If not, it would not be competitive with other non-parallel implementations. The algorithm inherits the
same approach of DCI-CLOSED. Recall that the projection used by DCI-CLOSED maintains the information regarding every singleton and this it does not have the side-effect of generation spurious itemsets. However, new obstacles arise due to parallelism.

There is a trade-off between the number of projections that can exist at the same time and the amount of memory we can use. Also the memory access patterns are affected. In principle different threads may work on different projections, thus resulting in increased memory requirements and possible cache pollution. On the other hand, forcing every thread to work on the same projection would introduce many implicit barriers that remarkably reduce the parallelism degree.

For what regards the strategies SS and SD, the thread that receives the job corresponding to the itemset \( \langle i_h, i_{h+1} \rangle \), which is the first needing the conditional bitmap \( vb_{i_h} \), is in charge of materializing the projection. The other threads will keep waiting until this is completed before mining the job corresponding to the itemsets \( \langle i_h, i_{h+2} \rangle, \langle i_h, i_{h+3} \rangle, \) etc.

In the case of the DD load partitioning strategy, we try to take advantage of the available degrees of freedom in the generation of new jobs. Whenever one thread becomes idle, it first tries to steal a new job from another working thread, thus working on the same dataset projection. If this is not possible, i.e. there is nothing to steal, than the thread may build a new projection. Until the construction of the new projection is not completed, the thread cannot be requested to split its workload. However, we permit to more than one thread to create new projections at the same time, thus implicitly parallelizing also the dataset projecting process.

6.4 Experimental evaluation

Unfortunately, large multi-core CPUs are still not available. In order to assess the scalability of our algorithm we used a large SMP cluster. The experiments were conducted on an “IBM SP Cluster 1600” machine which was kindly provided us by Cineca (http://www.cineca.it). The cluster contains 60 SMP nodes, each equipped with 8 processors and 16 GB of RAM. Each processor is an IBM PowerPC 5, having only one core enabled out of two. For our experiments we used on node only, thus having 8 processors available.

6.4.1 Scalability

We tested our three scheduling strategies on different datasets, whose characteristics are reported in Appendix A. These strategies were plugged into two different versions of the same algorithm: one of them does not exploit projections. This is because we wanted to test the additional overheads introduced by the projection of the dataset. We just reported mining times, since the time to read the dataset is not parallelized.
Accidents, no projections, min.supp. = 8.8%

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Speed Up</th>
<th>Accidents, no projections, min.supp. = 8.8%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>310 sec.s</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Accidents, with projections, min.supp. = 8.8%

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Speed Up</th>
<th>Accidents, with projections, min.supp. = 8.8%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>169 sec.s</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: Speedup factors as a function of the number of threads.
### 6.4. Experimental evaluation

#### Connect, no projections, min.supp. = 5.9%

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>222 sec.s</td>
</tr>
</tbody>
</table>

#### Connect, with projections, min.supp. = 5.9%

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>179 sec.s</td>
</tr>
</tbody>
</table>

Figure 6.3: Speedup factors as a function of the number of threads.
Figure 6.4: Speedup factors as a function of the number of threads.
Figure 6.5: Speedup factors as a function of the number of threads.
As expected, the entirely static strategy SS was the one resulting in the worst performance in every experiment conducted. In some tests, it hardly reaches a sufficient speedup: in the tests conducted with the USCensus1990 dataset it never reaches a speedup of 3. It is well known that splitting the visit of the lattice of frequent patterns result in very unbalanced sub-tasks: a few tasks may be much more expensive than all the others. Therefore, our SS strategy fails in well balancing the load among the various threads because it considers every sub-task having the same cost.

On the other hand, our two dynamic strategies provide a considerable improvement. According to the SD strategy, the algorithm partitions the search space in a quite large number of independent sub-tasks, that are self-scheduled by the pool of concurrent threads. Threads receiving low cost sub-tasks will ask for additional workloads, while a thread receiving a demanding sub-task will not be forced to fulfill other requests.

The SD strategy may fail in well balancing the load only when a very few sub-tasks have a huge cost. In this case it may happen that it is not possible to compensate these few demanding tasks with a large set of small-sized ones. As proved by our experiments, our lastly proposed strategy DD helps to alleviate this problem. Once one thread has completed his task, he eagerly steals work of some other running thread. Therefore, even if an initial assignment of work is very unbalanced, large workloads may be dynamically split into smaller ones executed on other threads. The improvement of DD becomes significant when projections are performed, i.e. when a more dynamic and adaptive support is required from the pool of threads.

We also report the results achieved on the Connect dataset, where none of the strategies proposed resulted in a performance less close to the ideal case. This is probably due to the additional synchronizations required by the DD strategy that do not allow to profit of its great adaptiveness. Further analysis are however needed to understand in depth this behavior.

Tough the results reported refer to a single minimum supports, in our experiments we witnessed an identical behavior for different support thresholds.

### 6.4.2 SIMD instruction set

The solutions adopted in this section do not regard the parallelization of DCIClosed, but rather the applicability of SIMD instructions to such algorithm. The same improvements are inherited by MT-Closed.

There are three precise points in DCI-Closed algorithm we are interested in. These are: tid-list intersection, order-preservation check and closure computation. The cost of these operations determine effectiveness of the algorithm. Our MT-Closed algorithm behaves as follows:

**tid-list intersection.** In line 18 of Alg. 5 the bit-vector representing the tid-list of the current order-preserving key-pattern K is intersected with the one of
the item $i$, in order to find the tid-list of the new candidate $Y = K \cup i$. The number of bits set in the resulting tid-lists corresponds the the support of the new itemset. There are a number of tricks for counting the number of bits set in a 32-bit memory location, the one we used exploits additions and logical operators for a total of 16 operations.

**order-preservation check.** According to the Extension Lemma, if there exist an item $j \in Y^-$ such that $g(Y) \subseteq g(j)$, then the $Y$ is not an order-preserving itemset and it must be discarded (see Alg. 5 line 20). The operation $a \subseteq b$ is translated into the couple of bit-wise operations: $(a \& b) == a$. Note that it is usually not necessary to scan the whole tid-list to discover that the inclusion does not hold.

**closure computation.** This is actually another application of the Extension Lemma. Alg. 5 in line 11 discovers all the items $j \in Y^+$ such that $g(Y) \subseteq g(j)$, that is $j \in c(Y)$. The same considerations for the order-preservation check hold in this case.

In [45] it is shown how to optimize these operations by reusing some information collected along the path of the lattice visit. However, they share the nice property of being very suitable for a SIMD approach. All these three basic operations are characterized by high spatial locality, and can effectively exploit the streaming instructions performing 128-bit operations that equip modern processor architectures, e.g. AMD 3DNow! and Intel SSE2 instruction sets. The Itanium 2 processor also includes the POPCNT instruction, that counts the number of bits set in a 64-bit long string in one clock cycle, and this instruction will probably be included in newly released processors supporting SSE5.

We implemented both of the above three functions using Intel SSE2 extensions that allow not only to load and store quad-words with a single instruction, but also to perform the simple operations we need directly on quad-words in only one CPU cycle. We can thus theoretically quadruplicate the throughput of each one of the above functions. For instance, we count the number of bits set in a 128 bit quad-word, still with only 16 operations. In general we do not expect to have a fourfold improvement in the performance of the algorithm, however the improvement in the mining time resulted to be significant on several datasets, as reported in Figure 6.4.2 which plots the relative speedups obtained with the exploitation of SSE2 instruction sets on a Intel Xeon processor.

Given this clear trend of integration of SIMD co-processors in modern CPU architectures, we think it is important to report on the exploitation of this interesting aspect.
6.5 Conclusions

We presented MT-Closed, the first multi-threaded parallel algorithm for extracting closed frequent itemsets from transactional databases. MT-Closed design was strongly inspired by nowadays trends in the field of processor architectures that indicate that the number of processor cores integrated in a single chip will continue to increase.

This algorithm is based on DCI-Closed because of two important features of its own. First, its peculiar duplicate checking method that allows an easy decomposition of the mining tasks into independent sub-tasks. Second, the vertical bitmap representation of the dataset allowed to exploit the streaming SIMD instructions that equip modern processor architectures. Finally, we took into account also the cache efficiency of the algorithm and the possibility to take advantage of memory hierarchies in a multi-threaded algorithm.

As a result of its accurate design, MT-Closed performances are impressive. In many experiment we measured quasi-linear speedups with a pool of 8 processors. These promising result allow us to predict that the proposed solutions would exploit efficiently also hardware architectures with a larger number of processors/cores.

![Figure 6.6: Speedup given by the exploitation of Intel’s SSE2 SIMD instructions.](image-url)
In this thesis we proposed several high performance algorithms for the extraction of closed frequent itemsets from transactional databases. Our interest in high performance algorithms is inspired by emerging computer architectures. Multi-core CPUs versus streaming GPU processors, complex cache memory hierarchies and SIMD instruction sets pose new challenges in the design of data mining algorithm. Parallel algorithms, careful memory management and access patterns, possibility to exploit SIMD instructions become mandatory issues in the designing of a modern algorithm.

We believe that high performance not only means to design fast algorithms but also to find new solutions to challenging problems.

In reviewing the state-of-the-art algorithms for frequent pattern mining, we discussed in detail some of the issues related to frequent pattern mining algorithm: computational complexity, data size and the opportunities provided by emerging computer architectures. We showed how these problems arise in every of those algorithms.

We also used our experience in constrained pattern mining to support this claim. We introduced CONQUEST \cite{10, 6}, a comprehensive software that helps the analyst during the discovery of interesting association rules. Interestingness is based on the constraints provided by the analyst. The mining engine of CONQUEST is based on a data-reduction approach. During each iteration of a level-wise algorithm, the properties of each constraint are used to prune the working dataset. We show that the resulting software is faster than other specialized algorithms, and in particular it deals with every of the high performance issue (except parallelism). More importantly, it is the first algorithm able to handle any conjunction of constraints at the same time.

For the remainder of the thesis, we focused on frequent closed itemsets. For each of the three aforementioned issues, we contributed a novel algorithm.

First we introduce DCI-CLOSED \cite{45}: a new algorithm for mining closed frequent itemsets. This algorithm outperforms other state-of-the-art algorithms. It faces the computational complexity issue thanks to an internal bitmap representation of the dataset that allows fast bitwise operations on tid-lists and provides high spatial locality of memory access patterns. Thanks to a novel duplicate detection technique, DCI-CLOSED is the only algorithm that can mine dense datasets without maintaining the collection of closed itemsets mined so far. It has other interesting features that allowed us to design both out-of-core and a parallel algorithms inspired by DCI-CLOSED.

OOC-CLOSED \cite{46} is the first algorithm for mining closed frequent itemsets in secondary memory. This problem was never solved before due to the difficulty of
identifying closed itemsets in a partition of the original dataset. We show that it is easy to create easy sub-problems that can be solved in main memory, and that the problem of detecting non closed itemsets possibly generated boils down the trivial problem of sorting strings in secondary memory. We thus solved the data size issue: we showed that OOC-CLOSED che mine very large datasets (about 2GB) using any given fixed amount of memory.

Finally, we proposed the first parallel algorithm for mining closed frequent itemsets called MT-CLOSED [47]. We show that it is possible to partition the mining problem into completely independent tasks that can be assigned to a pool of threads. We analyze different task assignment and data partitioning strategies, including a non trivial work-stealing strategy. This is particularly interesting since it allows a good load balancing even some mining subtasks are largely more expensive than the others. We thus dealt with the last issue providing an algorithm that is able to exploit the processing power of multiple cores. Also, we showed that MT-CLOSED can gain an additional speed-up when using SIMD instructions while executing operations on tid-lists.

The preset trend suggests that emerging computer architectures will become more complex over time, with tenths or hundreds of CPUs, complex cache hierarchies, and probably streaming and pipelined frameworks. With this work, we believe to give a contribution in the direction of harnessing modern computer architectures and, at the same time, to give new insights on the mining of closed patterns.
A

Description of datasets

A.1 Description of the datasets used in this thesis

Throughout this thesis we used a collection of publicly available datasets. Here we are going to describe their characteristics:

- **Chess** contains a set of chess moves
- **Mushroom** describes the physical characteristics of a set of mushrooms, including whether they are poisonous or not
- **Retail** collection of market baskets coming from a Belgian shop
- **Connect** contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced
- **Pumsb** contains census data for population and housing from PUMS (Public Use Microdata Sample)
- **Pumsb** contains census data for population and housing from PUMS (Public Use Microdata Sample)
- **Pumsb** the same as Pumsb without items having a support larger that 80%
- **Kosarak** contains (anonymized) click-stream data of a hungarian online news portal
- **Accidents** comes from a Belgian study on car accidents
- **USCensus1990** contains census information about a very small portion of U.S. population
- **WebDocs** comes from a large collection of web documents, each transaction consists in the term IDs a document contains
- **Synth2GB** is a large synthetic dataset produced by the IBM generator

In Table A.1 we report some statistics on these datasets. The table shows the number of distinct items, the number of transactions, the minimum, average and maximum transaction length, the size in KB, and finally the density defined as the number ratio of bits set in the bitmap representing the dataset.

Usually we distinguish between sparse and dense datatasets. The former contains loosely correlated items, and generally short patterns can be extrated from them. The latter contain highly correlated items, and a large number of long patterns is usually extracted even with a low minimum support threshold. Usually the density measure
Table A.1: Characteristics of the datasets used.

| Dataset     | $|I|$ | $|D|$ | $t_{\text{min}}$ | $t_{\text{avg}}$ | $t_{\text{max}}$ | size in KB | density |
|-------------|-----|-----|-----------------|-----------------|-----------------|------------|---------|
| Chess       | 75  | 3,196 | 37              | 37.0            | 37              | 334        | 0.493   |
| Mushroom    | 119 | 8,124 | 23              | 23.0            | 23              | 557        | 0.193   |
| Retail      | 16,470 | 88,162 | 1              | 10.3            | 76              | 4,069      | 0.0006  |
| Connect     | 129 | 67,557 | 43              | 43.0            | 43              | 9,038      | 0.333   |
| Pumsb*      | 2,088 | 49,046 | 49              | 50.5            | 63              | 11,027     | 0.024   |
| Pumsb       | 2,113 | 49,046 | 74              | 74.0            | 74              | 16,298     | 0.035   |
| Kosarak     | 41,270 | 990,002 | 1              | 8.1             | 2,497           | 31,278     | 0.0002  |
| Accidents   | 468 | 340,183 | 18              | 33.8            | 51              | 34,677     | 0.072   |
| USCensus1990 | 396 | 2,458,285 | 68              | 68.0            | 68              | 533,113    | 0.165   |
| WebDocs     | 5,267,656 | 1,692,082 | 1              | 177.2           | 71,472         | 1,447,158  | 0.00003 |
| Synth2GB    | 2,628 | 1,330,293 | 212             | 349.5           | 440             | 2,097,151  | 0.133   |

we reported is a good index of the dense-ness of the dataset. However it happens for a sparse dataset to have very dense sub regions.

The datasets we are considering range from very small (334KB) to very large ones (2GB), from very sparse (0.00003) to very dense ones (0.493). Note that the largest dataset is also the third most dense (0.133). We wanted to create a tough dataset from which it make sense to extract closed itemsets.

In fact another measure of the dataset density is the ratio between the number of frequent itemsets and the number of closed frequent itemsets. If ratio is one, than the dataset is sparse, there is low correlation between itemsets and it does not make sense to mine closed itemsets. Otherwise the dataset is dense.

We used FP-Close [28] to extract the frequent, closed frequent and maximal frequent itemsets from every dataset. For each dataset we plotted the itemsets length distribution. It turns out that USCensus1990, Pumsb and Pumsb* are dense dataset as expected since they contain census data. But also the Connect, Mushroom and Chess datasets are quite dense. The WebDocs dataset is very sparse, as espected due to its size and the differences among the objects it contains. And also the Accidents and Kosarak datasets are sparse. A slightly suprising result regards the Retail dataset. While we expected to be very sparse, as market basket datasets usually are, it seems to have interesting dense regions. While the number of closed itemsets is similar for very short patterns, it is significantly smaller than the number of frequent itemsets for length larger that 5.

Acutally we do not provide the plot about the Synth2GB dataset since FP-Close failed the mining tasks, even if it was run on a machine with about 2.5GB of memory.
A.1. Description of the datasets used in this thesis

Figure A.1: Itemsets length distributions on different datasets with different minimum support threshold.
A. Description of datasets

Figure A.2: Itemsets length distributions on different datasets with different minimum support threshold.


This is the nicest part of the thesis, and the easiest to write. When it comes to the acknowledgments, you realize that the years of hard work you have spent so far, have finally become a concrete and tangible object. At this point I realize that many people helped me during these three years.

First, I would like to thank my advisor Salvatore Orlando, and his colleague Raffaele, because they have been friends to me from my very first day in Pisa. I thank the crew of the High Performance Computing Laboratory, with whom it was a pleasure to work, and in particular Fabrizio, a friend rather than a colleague. I also thank Francesco, who worked with me at some of my first papers.

A special thank to Dr. Philip S. Yu and Michalis Vlachos, who hosted me at the IBM T.J. Watson Research Center. The three months I spent there were very interesting and fruitful. Unfortunately the two papers [49, 48] I wrote with them were slightly out of topic and could not be included in this thesis.

Ringrazio anche gli amici, al lavoro e non, a Pisa e lontano da Pisa. Un ringraziamento particolare va ai miei genitori e alla mia famiglia, ovvero alle persone che sempre mi hanno accompagnato e sempre mi saranno accanto. In fine, difficile trovare le parole, ringrazio la persona che rende le mie giornate serene ... Laura
List of PhD Theses

TD-2004-1 Moreno Marzolla
"Simulation-Based Performance Modeling of UML Software Architectures"

TD-2004-2 Paolo Palmerini
"On performance of data mining: from algorithms to management systems for data exploration"

TD-2005-1 Chiara Braghin
"Static Analysis of Security Properties in Mobile Ambients"

TD-2006-1 Fabrizio Furano
"Large scale data access: architectures and performance"

TD-2006-2 Damiano Macedonio
"Logics for Distributed Resources"

TD-2006-3 Matteo Maffei
"Dynamic Typing for Security Protocols"

TD-2006-4 Claudio Silvestri
"Distributed and Stream Data Mining Algorithms for Frequent Pattern Discovery"

TD-2007-1 Marco Giunti
"Secure Implementations of Typed Channel Abstractions"

TD-2007-2 Francesco Lelli
"Bringing Instruments to a Service-Oriented Interactive Grid"

TD-2007-3 Matteo Mordacchini
"Grid and Peer-to-Peer Resource Discovery Systems"

TD-2008-1 Claudio Lucchese
"High Performance Closed Frequent Itemsets Mining inspired by Emerging Computer Architectures"

TD-2008-2 Giulio Manzonetto
"Models and theories of lambda calculus"