# Generic Pattern Mining via Data Mining Template Library *

Mohammed J. Zaki, Nilanjana De, Feng Gao, Paolo Palmerini **, Nagender Parimi, Jeevan Pathuri, Benjarath Phoophakdee, and Joe Urban

Computer Science Department, Rensselaer Polytechnic Institute, Troy NY 12180

**Abstract.** Frequent Pattern Mining (FPM) is a very powerful paradigm for mining informative and useful patterns in massive, complex datasets. In this paper we propose the Data Mining Template Library, a collection of generic containers and algorithms for data mining, as well as persistency and database management classes. DMTL provides a systematic solution to a whole class of common FPM tasks like itemset, sequence, tree and graph mining. DMTL is extensible, scalable, and high-performance for rapid response on massive datasets. A detailed set of experiments show that DMTL is competitive with special purpose algorithms designed for a particular pattern type, especially as database sizes increase.

## 1 Introduction

Frequent Pattern Mining (FPM) is a very powerful paradigm which encompasses an entire class of data mining tasks, namely those dealing with extracting informative and useful patterns in massive datasets representing complex interactions between diverse entities from a variety of sources. These interactions may also span multiple-scales, as well as spatial and temporal dimensions. FPM is ideally suited for categorical datasets, which include text/hypertext data (e.g., news articles, web pages), semistructured and XML data, event or log data (e.g., network logs, web logs), biological sequences (e.g. DNA/RNA, proteins), transactional datasets, and so on. FPM techniques are able to extract patterns embedded in different subspaces within very high dimensional, massive datasets. FPM is very well suited to selecting or constructing good features in complex data and also for building global classification models of the datasets [26].

The specific tasks encompassed by FPM include the mining of increasingly complex and informative patterns, in complex structured and unstructured relational datasets, such as: Itemsets or co-occurrences [1] (transactional, unordered data), Sequences [2, 24] (temporal or positional data, as in text mining, bioinformatics), Tree patterns [25, 3] (XML/semistructured data), and Graph patterns [10, 13, 21, 22] (complex relational data, bioinformatics). Figure 1 shows

* This work was supported by NSF Grant EIA-0103708 under the KD-D program, NSF CAREER Award IIS-0092978, and DOE Early Career PI Award DE-FG02-02ER25538.

** The work was done while Paolo was at RPI

examples of these different types of patterns; in a generic sense a pattern denotes links/relationships between several objects of interest. The objects are denoted as nodes, and the links as edges. Patterns can have multiple labels, denoting various attributes, on both the nodes and edges.

The current practice in frequent pattern mining basically falls into the paradigm of incremental algorithm improvement and solutions to very specific problems. While there exist tools like MLC++ [12], which provides a collection of algorithms for classification, and Weka [20], which is a general purpose Java library of different data mining algorithms including itemset mining, these systems do not have an unifying theme or framework, there is little database support, and scalability to massive datasets is questionable. Moreover, these tools are not designed for handling complex pattern types like trees and graphs.

Our work seeks to address all of the above limitations. In this paper we describe Data Mining Template Library (DMTL), a generic collection of algorithms and persistent data structures, which follows a generic programming paradigm [4]. DMTL provides a systematic solution for the whole class of pattern mining tasks in massive, relational datasets. The main contributions of DMTL are as follows:

- The design and implementation of generic data structures and algorithms to handle various pattern types like itemsets, sequences, trees and graphs.
- Design and implementation of generic data mining algorithms for FPM, such as depth-first and breadth-first search.
- Persistent data structures for supporting efficient pattern frequency computations using a tightly coupled database (DBMS) approach.
- Native support for both a vertical and horizontal database formats for highly efficient mining.
- DMTL's support for pre-processing steps like data mapping and discretization of continuous attributes and creation of taxonomies. etc.

One of the main attractions of a generic paradigm is that the generic algorithms for mining are guaranteed to work for **any** pattern type. Each pattern has a list of properties it satisfies, and the generic algorithm can utilize these properties to speed up the mining. We conduct a detailed set of experiments to show the scalability and efficiency of DMTL for different pattern types like itemsets, sequences, trees and graphs. Our results indicate that DMTL is competitive with the special purpose algorithms designed for a particular pattern type, especially with increasing database sizes.

### 1.1 Related Work

Previous research in integrating mining and databases has mainly looked at SQL support. DMQL [8] is a mining query language to support common mining tasks. MSQL [9] is an extension of SQL to generate and selectively retrieve sets of rules from a large database. The MINE RULE SQL operator [15] and Query flocks [18] extend the semantics of association rules, allowing more generalized

queries to be performed. A comprehensive study of several architectural alternatives for database and mining integration were studied in [16]. This body of work is complementary to ours, since these SQL operators can be used as a front end to DMTL. Also, DMTL is optimized for the class of frequent patterns.

There has been limited work in integrating other mining tasks with databases. A middleware for classification was proposed in [5]; it decomposes and schedules classification primitives over a back-end SQL database. Two generic SQL operations called count-by-group (for class histograms) and compute-tuple-distances (for point distances) were identified in [6] for classification and clustering tasks, respectively.

## 2 Preliminaries

The problem of mining frequent patterns can be stated as follows: Let $\mathcal{N} = \{x_1, x_2, \ldots, x_{n_v}\}$ be a set of $n_v$ distinct nodes or vertices. A pair of nodes $(x_i, x_j)$ is called en edge. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_{n_l}\}$, be a set of $n_l$ distinct labels. Let $L_n : \mathcal{N} \rightarrow \mathcal{L}$, be a node labeling function that maps a node to its label $L_n(x_i) = l_j$, and let $L_e : \mathcal{N} \times cE \rightarrow \mathcal{L}$ be an edge labeling function, that maps an edge to its label $L_e(x_i, x_j) = l_k$.

A *pattern* $P$ is simply a relation on $\mathcal{N}$, $P \subseteq \mathcal{N} \times \mathcal{N}$, that is $P = \{(x_i, x_j) \mid x_i, x_j \in \mathcal{N}\}$, such that $P$ satisfies some user-specified conditions $\mathcal{C}$ (i.e., $\mathcal{C}(P)$ is true). It is also intuitive to represent a pattern $P$ as a graph $(P_V, P_E)$, with labeled vertex set $P_V \subset \mathcal{N}$ and labeled edge set $P_E = \{(x_i, x_j) \mid x_i, x_j \in P_V\}$. The number of nodes in a pattern $P$ is called its *size*. A pattern of size $k$ is called a $k$-pattern. In some applications $P$ is a symmetric relation, i.e., $(x_i, x_j) = (x_j, x_i)$ (unordered edges), while in other applications $P$ is anti-symmetric, i.e., $(x_i, x_j) \neq (x_j, x_i)$ (ordered edges). A path in $P$ is a set of distinct nodes $\{x_{i_0}, x_{i_1}, x_{i_n}\}$, such that $(x_{i_j}, x_{i_{j+1}})$ in an edge in $P_E$ for all $j = 0 \cdots n-1$. The number of edges gives the length of the path. If $x_i$ and $x_j$ are connected by a path of length $n$ we denote it as $x_i <_n x_j$. Thus the edge $(x_i, x_j)$ can also be written as $x_i <_0 x_j$.

Given two patterns $P$ and $Q$, we say that $P$ is a *subpattern* of $Q$ (or $Q$ is a super-pattern of $P$), denoted $P \preceq Q$ if and only if there exists a 1-1 mapping $f$ from nodes in $P$ to nodes in $Q$, such that for all $x_i, x_j \in P_V$: i) $L_n(x_i) = L_n(f(x_i))$, ii) $L_e(x_i, x_j) = L_e(f(x_i), f(X_j))$, and iii) $(x_i, x_j) \in P_V$ iff (if and only if) $(f(x_i), f(x_j)) \in Q_V$. In some cases we are interested in embedded subpatterns. $P$ is an *embedded subpattern* of $Q$ if: i) $L_n(x_i) = L_n(f(x_i))$, iii) $L_e(x_i, x_j) = L_e(f(x_i), f(X_j))$, and iii) $(x_i, x_j) \in P_V$ iff $f(x_i) <_l f(x_j)$, i.e., $f(x_i)$ is connected to $f(x_j)$ on some path. If $P \preceq Q$ we say that $P$ is contained in $Q$ or $Q$ contains $P$.

A database $\mathcal{D}$ is just a collection (a multi-set) of patterns. A database pattern is also called an *object*. Let $\mathcal{O} = \{o_1, o_2, \ldots, o_{n_o}\}$, be a set of $n_o$ distinct *object identifiers (oid)*. An object has a unique identifier, given by the function $O(d_i) = o_j$, where $d_i \in \mathcal{D}$ and $o_j \in \mathcal{O}$. The number of objects in $\mathcal{D}$ is given as $|\mathcal{D}|$.

The *absolute support* of a pattern $P$ in a database $\mathcal{D}$ is defined as the number of objects in $\mathcal{D}$ that contain $P$, given as $\pi^a(P, \mathcal{D}) = |\{P \preceq d \mid d \in \mathcal{D}\}|$. The

*(relative) support* of $P$ is given as $\pi(P, \mathcal{D}) = \frac{\pi^a(P,\mathcal{D})}{|D|}$. A pattern is *frequent* if its support is more than some user-specified minimum threshold, i.e., if $\pi(P, \mathcal{D}) \geq \pi^{\min}$. A frequent pattern is *maximal* if it is not a subpattern of any other frequent pattern. A frequent pattern is *closed* if it has no super-pattern with the same support. The frequent pattern mining problem is to enumerate all the patterns that satisfy the user-specified $\pi^{\min}$ frequency requirement (and any other user-specified conditions).

The main observation in FPM is that the sub-pattern relation $\preceq$ defines a partial order on the set of patterns. If $P \preceq Q$, we say that $P$ is more general than $Q$, or $Q$ is more specific than $P$. The second observation used is that if $Q$ is a frequent pattern, then all sub-patterns $P \preceq Q$ are also frequent. The different FPM algorithms differ in the manner in with they search the pattern space.

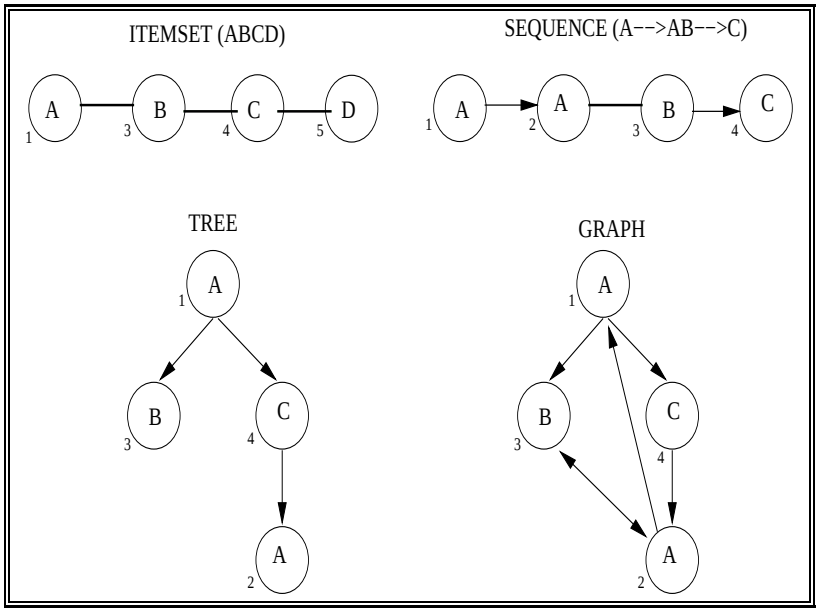


**Fig. 1.** FPM Instances

### 2.1 FPM Instances

Some common types of patterns include itemsets, sequences, trees, and graphs, as shown in Figure 1. In fact, every pattern can be modeled as a graph; the nodes ($x_i$) are shown under each circle and the node labels ($L_n(x_i)$) are shown inside the circle, whereas edge labels have been omitted.

In an itemset [1] no two nodes have the same label. Let $V = \{x_1, x_2, \cdots x_k\}$ be a node set such that $L_n(x_i) \neq L_n(x_j)$ for all $x_i, x_j \in V$, and $L_n(x_i) <$

$L_n(X_{i+1}$ for all $1 \leq i \leq k-1$. There are several possible formulation of the itemset pattern: i) *vertex-only*: An itemset pattern $P$ is just a of vertices, i.e., $P_V = V$ and $P_E = \emptyset$, ii) *linear*: Figure 1 shows another formulation, where the itemset is defined as $P_V = V$, and $P_E = \{(x_i, x_{i+1}) | x_i, x_{i+1} \in P_V\}$, iii) *clique*: A third alternative is to represent itemset $P$ as a clique, i.e., $P_V = V$ and $P_E = \{(x_i, x_j) \mid i < j \text{ and } x_i, x_j \in P_V\}$.

In sequence mining [2], a sequence is modeled as an ordered list of itemsets, and thus the different nodes in a sequence can have the same label. We can model a sequence pattern $P$ as being made up of a sequence of $n$ itemsets $P^i$, $i = 1, \cdots n$, using the linear formulation (as shown in Figure 1); note that using the vertex-only formulation is problematic, since it results in a disconnected pattern. Thus $P$ has a vertex set made up of $n$ disjoint subsets $P_V = \bigcup_{i=1}^{n} P_V^i$. The edge set $P$ contains all the edges within $P^i$ (consecutive and undirected), and $P_E$ contains, a directed edge for every pair of consecutive itemsets, i.e., from the last node of $P^i$ to the first node of $P^{i+1}$.

In tree mining [25, 3], typically rooted, ordered and labeled trees are considered. Thus a tree pattern $P$ consists of the vertex set $P_V = \{r, x_1, x_2, \cdots\}$, where $r$ is a special node called root. A tree pattern must satisfy all tree properties, namely i) the root has no parent, i.e., $(x_i, r) \notin P_E$ for any $x_i \in P_V$, ii) the edges are directed, i.e., if $(x_i, x_j) \in P_E$, then $(x_j, x_i) \notin P_E$), iii) a node has only one parent, i.e., if $(x_i, x_j) \in P_E$, then $(x_k, x_j) \notin P_E$ for any $x_k \neq x_i$, iv) the tree is connected, i.e., for all $x_i \in P_V$, there exists a path from the root $r$ to $x_i$, and v) tree has no cycles. Furthermore for ordered trees the order of a nodes' children matters. This means that there is an ordering of edges in $P_E$, such that $(x_i, x_j)$ comes before $(x_i, x_k)$ in $P_E$ only if $x_j$ is before $x_k$ in the ordering of $x_i$'s children.

Finally, by definition a pattern can model any general graph, as well as any special constraints that might appear in graph mining [10, 13, 21], such as connected graphs, or induced subgraphs. It is also possible to model other patterns such as DAGs (directed acyclic graphs).

## 3 DMTL: Data Structures and Algorithms

The C++ Standard Template Library (STL) provides efficient, generic implementations of widely used algorithms and data structures, which tremendously aid effective programming. Like STL, DMTL is a collection of generic data mining algorithms and data structures. In addition, DMTL provides persistent data and index structures for efficiently mining any type of pattern or model of interest. The user can mine custom pattern types, by simply defining the new pattern types, but there is no need to implement a new algorithm, since any generic DMTL algorithm can be used to mine them. Since the mined models and patterns are persistent and indexed, this means the mining can be done efficiently over massive databases, and mined results can be retrieved later from the persistent store.

Following the ideology of generic programming, DMTL provides a standardized, general, and efficient implementation of frequent pattern mining tasks by

isolating the concept of data structures or containers, as they are called in generic programming, from algorithms. DMTL provides container classes for representing different patterns (such as itemsets and sequences) and collection of patterns, containers for database objects (horizontal and vertical), and containers for temporary mining results. These container classes support persistency when required.

Generic algorithms, on the other hand are independent of the container and can be applied on any valid container. These include algorithms for performing intersections of the vertical lists [23–25] for itemsets or sequences or other patterns. Generic algorithms are also provided for mining itemsets and sequences [1, 17, 23, 24], as well as for finding the maximal or closed patterns [7, 27]. Finally DMTL provides support for the database management functionality, preprocessing support for mapping data in different formats to DMTL's native formats, as well as for data transformation (such as discretization of continuous values).

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

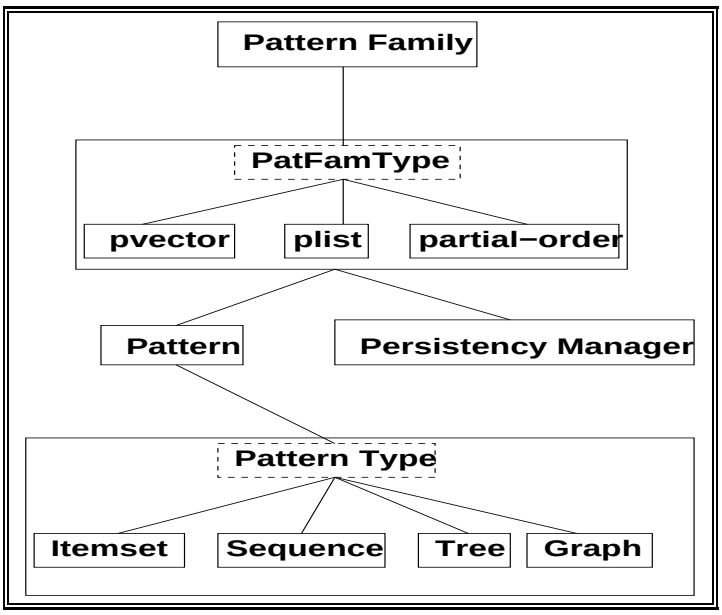


**Fig. 2.** DMTL Container Hierarchy

Figure 2 shows the different DMTL container classes for PMT and the relationship among them. At the lowest level set the different kinds of pattern-types one might be interested in mining (such as itemsets, sequences, and several

variants). A pattern is uses the base pattern-type classes to provide a generic container. There are several pattern family types (such as pvector, plist, etc.) which together with a persistency manager class make up different pattern family classes. More details on each class appears below.

### 3.1 Pattern

In DMTL a pattern is a generic container, which can be instantiated as an itemset, sequence, tree or a graph, specified as `Pattern<class P>` by means of a template argument called pattern-type (`P`). A generic pattern is simply a pattern-type whose frequency we need to determine in a larger collection or database of patterns of the same type.

### 3.2 Pattern Type

This allows users to select the type of pattern they want to mine, and as long as certain operations are defined on the pattern-type all the generic algorithms provided by DMTL can be used. The main source of flexibility of PMT is that developers can easily define new types of patterns to suit their needs and once the operations are defined on them all the generic algorithms of DMTL can be used on the new pattern types. For example, an itemset can be defined as pattern-type `vector<int>`, denoting a set of items (`int` in this case), A sequence pattern-type can defined as `list<vector<int>>`, denoting an ordered list of itemsets. If we want to include a time field along with the different itemsets in a sequence, we can define a new sequence type as follows `list<pair<time, vector<int>>>`, i.e., a list of (`time,vector<int>`) pairs, where time is a user-defined type to note when each event occurs. All algorithms are guaranteed to work with **any** pattern type.

### 3.3 Pattern Family

In addition to the basic pattern classes, most pattern mining algorithms operate on a collection of patterns. The pattern family is a generic container `PatternFamily<class PatFamType>` to store groups of patterns, specified by the template parameter `PatFamType`. `PatFamType` represents some persistent class provided by DMTL, that provides seamless access to the members, whether they be in memory or on disk. All access to patterns of a family is through the iterator provided by the `PatFamType` class. `PatternFamily` provides generic operations to add and remove patterns to/from the family, to find the maximal or closed patterns in the family, as well as a `count()` function that finds the support of all patterns, in the database, using functions provided by the database class.

### 3.4 Pattern Family Type

DMTL provides several persistent classes to store groups of patterns. Each such class is templatized on the pattern-type ($P$) and a persistency manager class `PM`. An example is `pvector<class P, class PM>`, a persistent vector class. It has the same semantics as a STL vector with added memory management and persistency. Thus a pattern family for itemsets can be defined in terms of the pvector class as follows: `PatternFamily<pvector<Itemset, PM>>`. Another class is `plist<P,PM>`. Instead of organizing the patterns in a linear structure like a vector or list, another persistent family type DMTL class, `partial-order<P,PM>`, organizes the patterns according to the sub-pattern/super-pattern relationship. While pvector and partial-order provide the same interface, certain operations will be more efficient in one class than the other. For example, inserts and deletions are cheaper for `plists`, while the maximality and closed testing functions will be cheaper for partial-orders, since the patterns are already organized according to sub/super-pattern relation.

### 3.5 Persistent Containers

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

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

### 3.6 Generic Mining Algorithms

The pattern mining task can be viewed as a search over the pattern space looking for those patterns that match the minimum support constraint. For instance in itemset mining, the search space is the set of all possible subsets

of items. Various search strategies are possible leading to several popular variants of the mining algorithms. DMTL provides generic algorithms encapsulating these search strategies; by their definition these algorithms can work on any type of pattern: Itemset, Sequence, Tree or Graph. An example is the generic algorithm `DFS-Mine<class PatFamType> (PatternFamily<PatFamType> &pf, DB &db, ...)` , which mines the frequent patterns using a depth-first search (DFS) [23, 24]. The generic DFS mining algorithm takes in a pattern family and the database. The types of patterns and persistency manager are specified by the pattern family type. The DFS algorithm in turn relies on other generic subroutines for creating equivalence classes, for generating candidates, and for support counting. There is also a generic `BFS-Mine` that performs Breadth-First Search [1, 17] over the pattern space.

## 4 DMTL: Persistency & Database Support

DMTL is the back-end server that actually provides the persistency, and indexing support for both the patterns and the database. DMTL supports DMTL by seamlessly providing support for memory management, data layout, high-performance I/O, as well as tight integration with database management systems (DBMS). It supports multiple back-end storage schemes including flat files, embedded databases, and relational or object-relational DBMS. DMTL also provides persistent pattern management facilities, i.e., mined patterns can themselves be stored in a pattern database for retrieval and interactive exploration.

DMTL provides native database support for both the horizontal [1] and vertical [23–25] data formats. In the horizontal approach, each object has an oid along with the itemset comprising the object. Thus object with $oid = 1$ is the set $\{A, C, T, W\}$. In contrast, the vertical format maintains for each label (and itemset) its oid list, a set of all oids where it occurs. For example, the label $A$ appears in oids 1, 3, 4, and 5. Thus its oid list is given as 1345 (omitting set notation).

It is also worth noting that since in many cases the database contains the same kind of objects as the patterns to be extracted (i.e., the database can be viewed as a pattern family), the same database functionality used for horizontal format can be used for providing persistency for pattern families. It is relatively straightforward to store a horizontal format object, and by extension, a family of such patterns, in any object-relational database. Thus the persistency manager for pattern families can handle both the original database and the patterns that are generated while mining. DMTL provides the required buffer management so that the algorithms continue to work regardless of whether the database/patterns are in memory or on disk.

### 4.1 Vertical Attribute Tables

To provide native database support for objects in the vertical format, DMTL adopts a fine grained data model, where records are stored as *Vertical Attribute*

*Tables* (VATs). Given a database of objects, where each object is characterized by a set of properties or attributes, a VAT is essentially the collection of objects that share the same values for the attributes. For example, for a relational table, `cars`, with the two attributes, `color` and `brand`, a VAT for the property `color=red` stores all the transaction identifiers of cars whose color is red. The main advantage of VATs is that they allow for optimizations of query intensive applications like data mining where only a subset of the attributes need to be processed during each query. As was mentioned earlier these kinds of vertical representations have proved to be useful in many data mining tasks [23–25].

In DMTL there is typically one VAT per pattern. A VAT is an entity composed of a *body*, which contains the list of object identifiers in which a given pattern occurs. For storing database sequences a VAT needs, in addition, a `time` field for each occurrence of the pattern. For tree and graph patterns the body type is different. A VAT is defined as `VAT<class V>`, where `V` is a vat-type class.

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

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

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

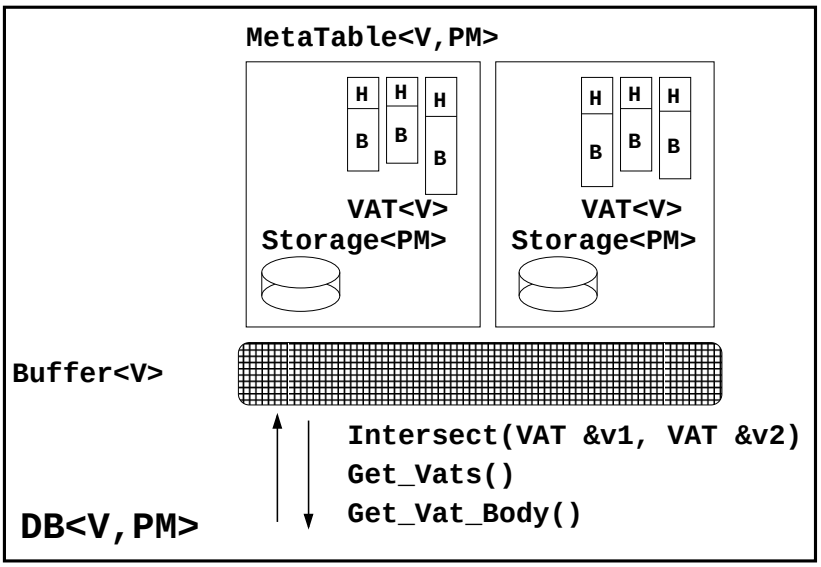


**Fig. 3.** DMTL: High level overview of the different classes used for Persistency

### 4.2 Storage & Persistency Manager

The database support for VATs and for the horizontal family of patterns is provided by DMTL in terms of the following classes, which are illustrated in Figure 3.

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

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

`Storage<class PM>:` The generic persistency-manager class that implements the physical persistency for VATs and other classes. The class PM provides the actual implementations of the generic operations required by Storage. For example, PM_metakit and PM_gigabase are two actual implementations of the `Storage` class in terms of different DBMS like Metakit [19], a persistent C++ library that natively supports the vertical format, and Gigabase [11], an object-relational database. Other implementations can easily be added as long as they provide the required functionality.

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

`DB<class V, class PM>:` The database class which holds a collection of `Metatables`. This is the main user interface to VATs and constitutes the database class

DB referred to in previous sections. It supports VAT operations such as intersection, as well as the operations for data import and export. The double template follows the same format as that of the `Metatable` class.

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

A diagram of the class interaction is displayed in Figure 3. As previously stated, the `DB` class is the main DMTL interface to VATs and the persistency manager for patterns. It has as data members an object of type `Buffer<V>` and a collection of `MetaTables<V,PM>`.

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

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

### 4.3 VAT Persistency

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

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

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

### 4.4 Buffer Management

The `Buffer` class provides methods to access and to manage a fixed size buffer where the most recently used VATs/patterns are stored for fast retrieval. The

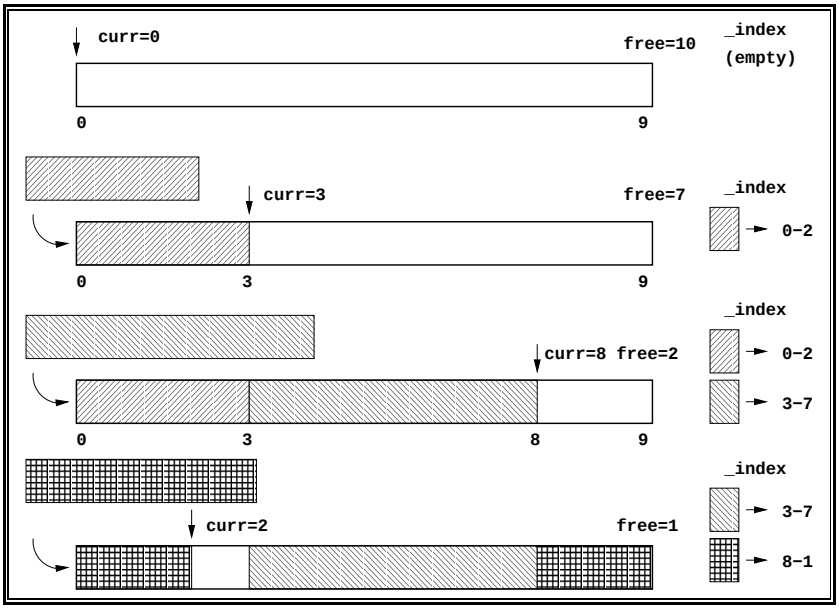


**Fig. 4.** Buffer Management: The buffer manager keeps track of the current position in the buffer (curr), the amount of free space (free), and an index to keep track of where an object is. As new objects are inserted these are updated and when an existing object is replaced when the buffer becomes full.

idea behind the buffer management implemented in the `Buffer` class is illustrated in Figure 4.

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

### 4.5 Storage

Physical storage of VATs and pattern families can be implemented using different storage systems, such as a DBMS or ad-hoc libraries. In order to abstract the details of the actual system used, all storage-related operations are provided in a generic class, `Storage`. Implementations of the `Storage` class for MetaKit [19] and Gigabase [11] backends are provided in the DMTL. Other implementations can easily be added as long as they provide the required functionality.

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

## 5 DMTL: Pre-processing Support

DMTL provides support for dynamic mapping of data into VATs at run-time over the same base database using a *mapping* class, which transforms original attribute values into *mapped values* according to a set of user specified mapping directives, contained in a configuration file. For every input database there has to be an XML configuration file. For the definition of the syntax of such file, we follow the approach presented in [14] by Talia *et al.*. The format of such file is the following.

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

### 5.1 Attributes

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

### 5.2 Mapping

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

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

In this case the field `price` will be mapped to `(max-min)/step = (5-1)/.5 = 10` values, labeled with integers starting from `0`. It is also possible to specify non-uniform discretizations, omitting the `step` attribute and explicitly specifying all the ranges and labels. For categorical values we can also specify a mapping, that allows for taxonomies or other groupings.

## 6 Experiments

DMTL is implemented using C++ Standard Template Library [4]. We present some experimental results on the time taken by DMTL to load databases and to perform different types of pattern mining on them. We used the IBM synthetic database generator [1] for itemset and sequence mining, the tree generator from [25] for tree mining and the graph generator by [13], with sizes ranging from $10k$ to $1000k$ (or 1 million). The experiment were run on a Pentium4 2.8Ghz Processor with 6GB of memory, running Linux.

Figure 5 shows the DMTL mining time versus the specialized algorithms for itemset mining (ECLAT [23]), sequences (SPADE [24]), trees (TreeMiner [25]) and graphs (gSpan [21]). For the DMTL algorithms, we show the time with a flat-file (Flat) persistency manager/database, with the metakit backend (Metakit) and the gigabase backend (Gigabase). The left hand column shows the effect of minimum support on the mining time for the various patterns. We find that for all pattern types DMTL is within a factor of 10 of the specialized algorithms even as we decrease the minimum support on a database with 100K records. The column on the right hand size shows the effect of increasing database sizes on these algorithms. We find that as the number of objects increase the gap between DMTL algorithms and the specialized ones starts to decrease. We expect that as we increase the number of records, the specialized algorithms will break down, while DMTL will continue to run since it explicitly manages the memory buffers. Comparing the three backend implementations, we find that the flatfile approach has a slight edge, but the object-oriented gigabase database backend is almost as fast. On the other hand the embedded database Metakit is generally slower.

Figure 6 shows the time taken to convert the input data into VATs. The times are shown for the three different backends (flat, metakit and gigabase) for upto 1 million objects. We find that these three approaches are roughly the same, with the maximum difference being a factor of 2.

## 7 Conclusions

In this paper we describe the design and implementation of the DMTL prototype for an important subset of FPM tasks, namely mining frequent itemsets, sequences, trees, and graphs. Following the ideology of generic programming, DMTL provides a standardized, general, and efficient implementation of frequent pattern mining tasks by isolating the concept of data structures or containers, from algorithms. DMTL provides container classes for representing different patterns, collection of patterns, and containers for database objects (horizontal and

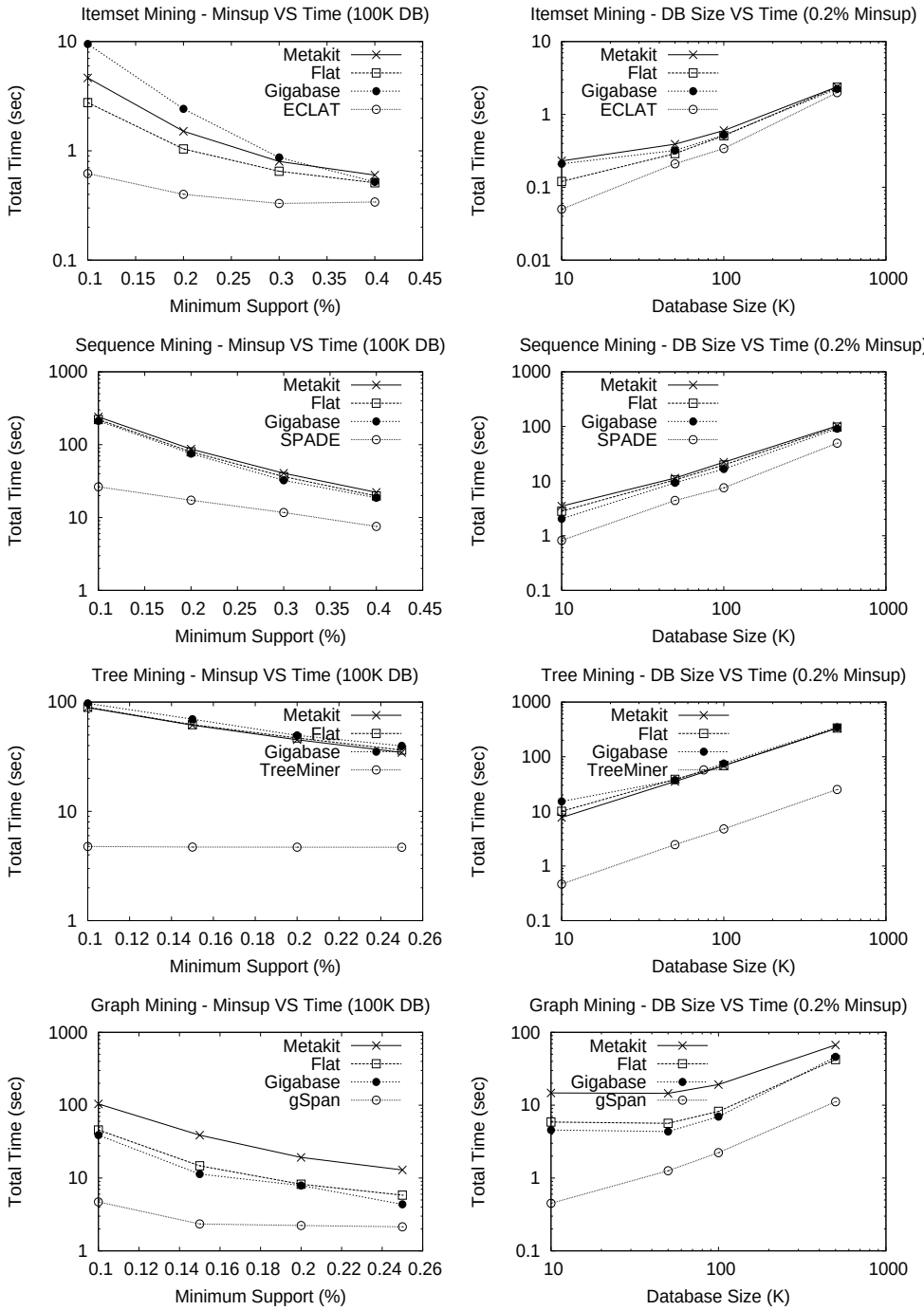


**Fig. 5.** Itemset, Sequence, Tree and Graph Mining: Effect of Minimum Support and Database Size

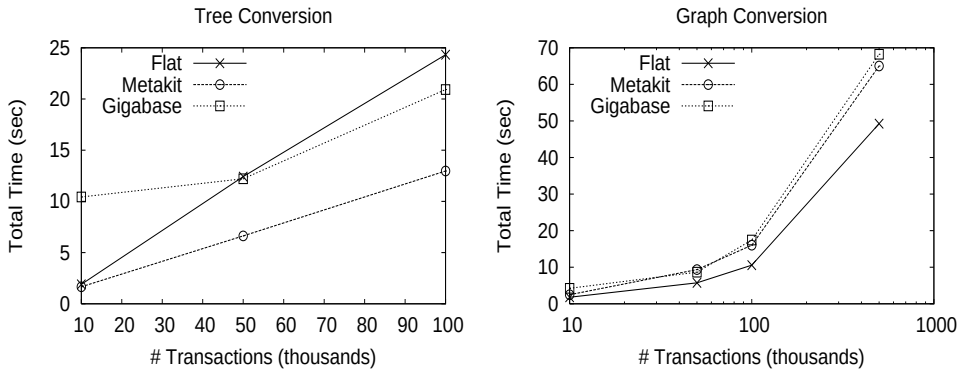


**Fig. 6.** Database Conversion and Loading Times

vertical). Generic algorithms, on the other hand are independent of the container and can be applied on any valid pattern. These include algorithms for performing intersections of the VATs, or for mining.

The generic paradigm of DMTL is a first-of-its-kind in data mining, and we plan to use insights gained to extend DMTL to other common mining tasks like classification, clustering, deviation detection, and so on. Eventually, DMTL will house the tightly-integrated and optimized primitive, generic operations, which serve as the building blocks of more complex mining algorithms. The primitive operations will serve all steps of the mining process, i.e., pre-processing of data, mining algorithms, and post-processing of patterns/models. Finally, we plan to release DMTL as part of open-source, and the feedback we receive will help drive more useful enhancements. We also hope that DMTL will provide a common platform for developing new algorithms, and that it will foster comparison among the multitude of existing algorithms.

## References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.
2. R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Intl. Conf. on Data Engg.*, 1995.
3. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *2nd SIAM Int'l Conference on Data Mining*, April 2002.
4. M. H. Austern. *Generic Programming and the STL.* Addison Wesley Longman, Inc., 1999.
5. S. Chaudhri, U. Fayyad, and J. Bernhardt. Scalable classification over SQL databases. In *15th IEEE Intl. Conf. on Data Engineering*, March 1999.
6. A. Freitas and S. Lavington. *Mining very large databases with parallel processing.* Kluwer Academic Pub., Boston, MA, 1998.

7. K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.
8. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. DMQL: A data mining query language for relational databases. In *1st ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1996.
9. T. Imielinski and A. Virmani. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery: An International Journal*, 3:373–408, 1999.
10. A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *4th European Conference on Principles of Knowledge Discovery and Data Mining*, September 2000.
11. Konstantin Knizhnik. Gigabase, object-relational database management system. http://sourceforge.net/projects/gigabase.
12. R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using mlc++, a machine learning library in c++. *International Journal of Artificial Intelligence Tools*, 6(4):537–566, 1997.
13. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.
14. C. Mastroianni, D. Talia, and P. Trunfio. Managing heterogeneous resources in data mining applications on grids using xml-based metadata. In *Proceedings of The 12th Heterogeneous Computing Workshop*, 2002.
15. R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *22nd Intl. Conf. Very Large Databases*, 1996.
16. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternatives and implications. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.
17. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.
18. D. Tsur, J.D. Ullman, S. Abitboul, C. Clifton, R. Motwani, and S. Nestorov. Query flocks: A generalization of association rule mining. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.
19. Jean-Claude Wippler. Metakit. http://www.equi4.com/metakit/.
20. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 1999.
21. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE Int'l Conf. on Data Mining*, 2002.
22. X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, August 2003.
23. M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, May-June 2000.
24. M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1/2):31–60, Jan/Feb 2001.
25. M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, July 2002.
26. M. J. Zaki and C.C. Aggarwal. Xrules: An effective structural classifier for xml data. In *9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, August 2003.
27. M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data Mining*, April 2002.