# Frequent Pattern Queries for Flexible Knowledge Discovery


Francesco Bonchi[1], Fosca Giannotti[1], and Dino Pedreschi[2]

[1] ISTI - CNR, Area della Ricerca di Pisa, Via Giuseppe Moruzzi, 1 - 56124 Pisa, Italy
[2] Dipartimento di Informatica, Via F. Buonarroti 2, 56127 Pisa, Italy



**Abstract.** In this paper we study data mining query language and optimizations in the context of a Logic-based Knowledge Discovery Support Environment. i.e., a flexible knowledge discovery system with capabilities to obtain, maintain, represent, and utilize both induced and deduced knowledge. In particular, we focus on frequent pattern queries, since this kind of query is at the basis of many mining tasks, and it seems appropriate to be encapsulated in a knowledge discovery system as a primitive operation. We introduce an inductive language for frequent pattern queries, which is simple enough to be highly optimized and expressive enough to cover the most of interesting queries. Then we define an optimized constraint-pushing operational semantics for our inductive language. This semantics is based on a frequent pattern mining operator, which is able to exploit as much as possible the given set of constraints, and which can adapt its behavior to the characteristics of the given input set of data.


## 1 Introduction

Knowledge Discovery in Databases is a complex iterative and interactive process which involves many different tasks that can bring the analyst from raw dirty data to actionable knowledge. A rigorous user interaction during such process is needed in order to facilitate efficient and fruitful knowledge manipulation and discovery. Such a rigorous interaction can be achieved by means of a set of *data mining primitives*, that should include the specification of the source data, the kind of knowledge to be mined, background or domain knowledge, interestingness measures for patterns evaluation, and finally the representation of the extracted knowledge. Providing a query language capable to incorporate all these features may result, like in the case of relational databases, in a high degree of expressiveness in the specification of data mining tasks, a clear and well-defined separation of concerns between logical specification and physical implementation of data mining tasks, and easy integration with heterogeneous information sources.

Clearly the implementation of this vision presents a great challenge. A path to this goal is indicated in [10] where Mannila introduces an elegant formalization for the notion of interactive mining process: the term *inductive database* refers to a relational database plus the set of all sentences from a specified class of sentences that are true of the data.

**Definition 1.** Given an instance $\mathbf{r}$ of a relation $\mathbf{R}$, a class $\mathcal{L}$ of sentences (patterns), and a selection predicate $q$, a pattern discovery task is to find a theory

$$\mathcal{T}h(\mathcal{L}, \mathbf{r}, q) = \{s \in \mathcal{L} | q(\mathbf{r}, s) \text{ is true}\}$$

The selection predicate $q$ indicates whether a pattern $s$ is considered interesting, and it is defined as a conjunction of *constraints* defined by the analyst. In other words, the inductive database is a database framework which integrates the raw data with the knowledge extracted from the data and materialized in the form of patterns. In this way, the knowledge discovery process consists essentially in an iterative querying process, enabled by a query language that can deal either with raw data or patterns.

### 1.1 Logic-based Knowledge Discovery Support Environment

The notion of inductive database fits naturally in rule-based languages, such as *deductive databases* [6]. A deductive database can easily represent both extensional and intensional data, thus allowing a higher degree of expressiveness than traditional relational algebra. Such capability makes it viable for suitable representation of domain knowledge and support of the various steps of the knowledge discovery process.

The last consideration leads to the definition of a *Logic-based Knowledge Discovery Support Environment* (LKDSE in the following) as a deductive database programming language equipped with inductive rules. The main idea of the previous definition is that of providing a simple way for modelling the key aspects of a data mining query language:

- the source data and the background knowledge are represented by the relational extensions;
- deductive rules provide a way of integrating background knowledge in the discovery process, pre-processing source data, post-processing and reasoning on the newly extracted knowledge;
- inductive rules provide a way of declaratively invoking mining procedures with explicit representation of interestingness measures.

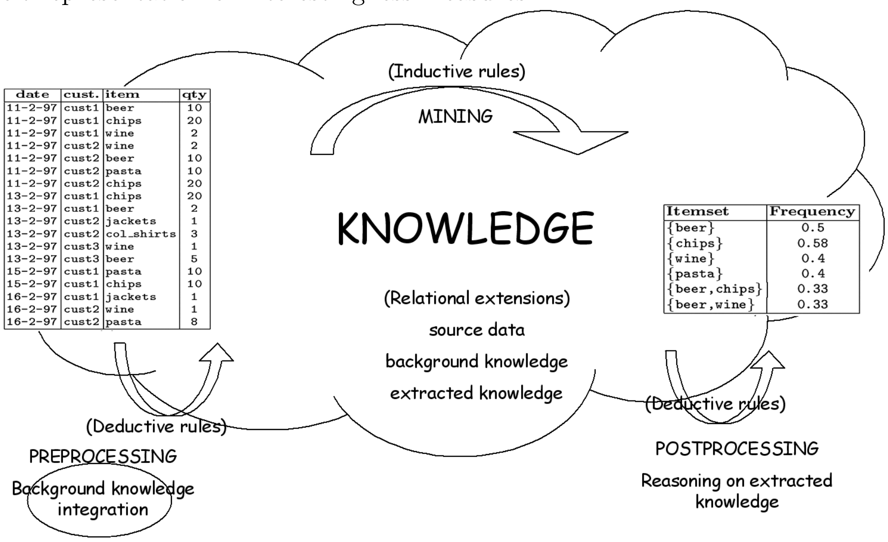


**Fig. 1.** The vision of Logic-based Knowledge Discovery Support Environment.

The main problem of a deductive approach is how to choose a suitable representation formalism of the inductive part, capable of expressing the correspondence between the deductive part and the inductive part. More specifically, the problem is how to formalize the specification of the set $\mathcal{L}$ of patterns in a way such that each pattern $s \in \mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$ is represented as an independent (logical) entity (i.e., a predicate) and each manipulation of $\mathbf{r}$ results in a corresponding change in $s$.

A first attempt to define inductive rules on a deductive database is in [6]. In this work the notion of inductive rules in a deductive framework is elegantly defined by means of *user-defined aggregates* on the Datalog++ logic-based database language and its practical implementation, namely the $\mathcal{LDL}$++ deductive database system. The resulting LKDSE has been named $\mathcal{LDL}$-$\mathcal{M}$ine. For lack of space, we shall omit a presentation of $\mathcal{LDL}$++, and confine ourselves to mention that it is a rule-based language with a Datalog-like syntax, and a semantics that extends that of relational database query languages with recursion [7].

The main drawback of using user defined aggregates as a mean to define inductive queries is the atomicity of the aggregate that makes us loose optimization opportunities. For instance, in frequent pattern discovery a user may wish to mine only frequent patterns or rules that satisfy some constraints. This constraints could be used to reduce the search space of the computation as shown in many works in literature [11, 9, 5, 3, 4]. Unluckily in this approach such constraints can not be directly exploited by the inductive rules but can only be checked by deductive rules after the extraction of the frequent itemsets. In order to exploit constraints to reduce the search space of the mining algorithm one should redefine the mining aggregate for any particular constraint and query. This requires nontrivial programming effort for the analyst and however, constraints satisfaction would be behind the query level, thus losing the transparency which is one main requirement of inductive database.

### 1.2 Our Position and Objective

The objective of this research is to study *pattern discovery queries* (or *mining queries* or *inductive queries*), both by the point of view of the language and of the optimizations [2], in the context of a Logic-based Knowledge Discovery Support Environment based on Datalog++ as deductive database language, equipped with inductive rules.

In our vision, the data analyst should have a high-level vision of the data mining system, without worrying about the details of the computational engine, in the very same way a database designer has not to worry about query optimizations. The analyst must be provided with a set of primitives to be used to communicate with the data mining system, using a data mining query language. The analyst just needs to declaratively specify in the data mining query how the desired patterns should look like and which conditions they should obey (a set of constraints). Indeed, the task of composing all constraints and producing the most efficient mining strategy (execution plan) for the given data mining query should be left to an underlying *query optimizer*. This is the paradigm of *Declarative Mining*. Following this vision, we define a new language of inductive queries on top of a deductive database. In our framework an inductive rule is simply a conjunction of sentences about the desired patterns:

- These sentences are taken form a specified class of sentences and they can be seen as mining primitives, computed by a specialized algorithm (and not by aggregates as in the previous approach).
- The set of all admissible sentences is just some "syntactic sugar" on top of an algorithm. The algorithm is the *inductive engine*.
- Each sentence can be defined over some deductive predicates (relations) defining the data source for the mining primitive.

Therefore, in this setting we have a clear distinction between what must be computed by deductive engine (deductive rules) and what by the inductive engine (inductive rules). Moreover we clearly specify the relationship between the inductive and the deductive part of an inductive database: the deductive engine feeds the inductive engine with data sources by means of the deductive predicates contained in the inductive sentences; the inductive engine returns in the deductive database predicates representing patterns that satisfy all the sentences in the inductive rule. Having a well defined and restricted set of possible sentences allows us to write highly optimized algorithms to compute inductive rules.

In this paper, we focus on *frequent pattern queries* over a transactional database [1, 8], i.e. queries which model the pattern discovery task $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$, where $\mathcal{L}$ is the pattern domain of itemsets, and $q$ is a minimal frequency constraint, or in other words, $q$ selects itemsets which which appear in the transactional database a number of times greater than a user-defined minimum frequency threshold.

The rationale behind this choice is that this kind of query is a simple primitive operation (nothing more than counting) which is at the basis of practically every mining tasks, but which is usually the most time-consuming operation in any mining session. Therefore, it seems appropriate to encapsulate frequency counting in a knowledge discovery system as a primitive operation, and to study its optimizations.

## 2 Frequent Pattern Mining

**Definition 2 (Frequent Pattern Mining).** Let $\mathcal{I} = \{x_1, ..., x_n\}$ be a set of distinct literals, usually called *items*, where an item is an object with some predefined attributes (e.g., price, type, etc.). An *itemset* $X$ is a non-empty subset of $\mathcal{I}$. If $|X| = k$ then $X$ is called a *k-itemset*. A constraint on itemsets is a function $\mathcal{C} : 2^{\mathcal{I}} \rightarrow \{true, false\}$. We say that an itemset $I$ satisfies a constraint if and only if $\mathcal{C}(I) = true$. We define the *theory* of a constraint as the set of itemsets which satisfy the constraint: $Th(\mathcal{C}) = \{X \in 2^{\mathcal{I}} \mid \mathcal{C}(X)\}$. A *transaction database* $\mathcal{D}$ is a bag of itemsets $t \in 2^{\mathcal{I}}$, usually called *transactions*. The *cover* of an itemset $X$ in database $\mathcal{D}$, is the set of transactions in $\mathcal{D}$ which are superset of $X$: $cov_{\mathcal{D}}(X) = \{t \in \mathcal{D} \mid t \supseteq X\}$. The *support* of an itemset $X$ in database $\mathcal{D}$, denoted $supp_{\mathcal{D}}(X)$, is the cardinality of $cov_{\mathcal{D}}(X)$. Given a user-defined *minimum support* $\sigma$, an itemset $X$ is called *frequent* in $\mathcal{D}$ if $supp_{\mathcal{D}}(X) \geq \sigma$. This defines the minimum frequency constraint: $\mathcal{C}_{freq[\mathcal{D},\sigma]}(X) \Leftrightarrow supp_{\mathcal{D}}(X) \geq \sigma$. When the dataset and the minimum support threshold are clear from the context, we indicate the frequency constraint simply $\mathcal{C}_{freq}$. Thus with this notation, the *frequent itemsets mining problem* requires to compute the set of all frequent itemsets $Th(\mathcal{C}_{freq})$. In general given a conjunction of constraints $\mathcal{C}$ the *constrained frequent itemsets mining problem* requires to compute $Th(\mathcal{C}_{freq}) \cap Th(\mathcal{C})$.

*Example 3 (Market Basket Analysis).* The most natural way to think about a transaction database is the *sales database* of a retail store, where the content of each basket appearing at the cash register is recorded. In this context a transaction $\langle tid, X\rangle$ represents a basket identifier and its content. A transaction database can be represented also as a relational table as shown in Figure 2(a). In that table a transaction or basket identifier is not explicitly given, but for instance, one could use the couple $(\texttt{date}, \texttt{cust})$

| date | cust. | item | qty |
|---|---|---|---|
| 11-2-97 | cust1 | beer | 10 |
| 11-2-97 | cust1 | chips | 20 |
| 11-2-97 | cust1 | wine | 2 |
| 11-2-97 | cust2 | wine | 2 |
| 11-2-97 | cust2 | beer | 10 |
| 11-2-97 | cust2 | pasta | 10 |
| 11-2-97 | cust2 | chips | 20 |
| 13-2-97 | cust1 | chips | 20 |
| 13-2-97 | cust1 | beer | 2 |
| 13-2-97 | cust2 | jackets | 1 |
| 13-2-97 | cust2 | col_shirts | 3 |
| 13-2-97 | cust3 | wine | 1 |
| 13-2-97 | cust3 | beer | 5 |
| 15-2-97 | cust1 | pasta | 10 |
| 15-2-97 | cust1 | chips | 10 |
| 16-2-97 | cust1 | jackets | 1 |
| 16-2-97 | cust2 | wine | 1 |
| 16-2-97 | cust2 | pasta | 8 |
| 16-2-97 | cust3 | chips | 20 |
| 16-2-97 | cust3 | col_shirts | 3 |
| 16-2-97 | cust3 | brown_shirts | 2 |
| 18-2-97 | cust1 | pasta | 5 |
| 18-2-97 | cust1 | wine | 1 |
| 18-2-97 | cust1 | chips | 20 |
| 18-2-97 | cust1 | beer | 10 |
| 18-2-97 | cust2 | beer | 12 |
| 18-2-97 | cust2 | beer | 10 |
| 18-2-97 | cust2 | chips | 20 |
| 18-2-97 | cust2 | chips | 20 |
| 18-2-97 | cust3 | pasta | 10 |

(a)

| date | cust. | itemset |
|---|---|---|
| 11-2-97 | cust1 | {beer,chips,wine} |
| 11-2-97 | cust2 | {wine,beer,pasta,chips} |
| 13-2-97 | cust1 | {chips,beer} |
| 13-2-97 | cust2 | {jackets,col_shirts} |
| 13-2-97 | cust3 | {wine,beer} |
| 15-2-97 | cust1 | {pasta,chips} |
| 16-2-97 | cust1 | {jackets} |
| 16-2-97 | cust2 | {wine,pasta} |
| 16-2-97 | cust3 | {chips,col_shirts,brown_shirts} |
| 18-2-97 | cust1 | {pasta,wine,chips,beer} |
| 18-2-97 | cust2 | {beer,chips} |
| 18-2-97 | cust3 | {pasta} |

(b)

| item | price | type |
|---|---|---|
| beer | 10 | beverage |
| chips | 3 | snack |
| wine | 20 | beverage |
| pasta | 2 | food |
| jackets | 100 | clothes |
| col_shirt | 30 | clothes |
| brown_shirt | 25 | clothes |

(c)

**Fig. 2.** (a) A sample `sales` table and (b) one of its transactional representations; (c) the `product` table.

to indicate it. If our relational language allows set-valued fields, we can have the same relation in its transactional representation as in Figure 2(b).

In classical frequent pattern mining, the popular Apriori algorithm [1] exploits an interesting property of frequency for pruning the exponential search space of the problem: whenever the support of an itemset violates the frequency constraint, then all its supersets can be pruned away from the search space, since they will violate the frequency constraint too. This property of frequency is called *antimonotonicity* (see Definition 4) and is the basis of the breadth-first level-wise (from small itemsets to large itemsets) Apriori exploration and pruning of the search space.

**Definition 4 (Antimonotone constraint).** Given an itemset $X$, a constraint $\mathcal{C}_{AM}$ is *antimonotone* if $\forall Y \subseteq X : \mathcal{C}_{AM}(X) \Rightarrow \mathcal{C}_{AM}(Y)$.

As already stated frequency is clearly an antimonotone constraint. Many other kind of constraints with the same nice property can be defined. For instance one could be interested in mining frequent itemsets with a total sum of prices $\leq$ 50\$: this constraint is antimonotone because any itemset that already has a sum of prices greater than 50\$ will never produce a solution. Adding more items to the itemset will simply make it more expensive, so it will never satisfy the constraint. Such constraints can be pushed deeply down into the frequent pattern mining computation since they behave exactly as the frequency constraint: if they are not satisfiable at an early level (small patterns), they have no hope of becoming satisfiable later (larger patterns). Moreover, since any conjunction of antimonotone constraints is antimonotone as well, they can be exploited all together, in the same way of frequency, to prune the search space. The more antimonotone constraints the user specifies, the more selective the search will be.

The case is more subtle for constraints which exhibit the opposite property to antimonotonicity.

**Definition 5 (Monotone constraint).** Given an itemset $X$, a constraint $\mathcal{C}_M$ is *monotone* if: $\forall Y \supseteq X : \mathcal{C}_M(X) \Rightarrow \mathcal{C}_M(Y)$.

Since the frequency computation moves from small to large patterns, we can not push monotone constraints directly in it. At an early stage, if an itemset is too small or too cheap to satisfy a monotone constraint, we can not yet say nothing about its supersets. Perhaps, just adding a very expensive single item to the itemsets could raise the total sum of prices over the given threshold, thus making the resulting itemset satisfy the monotone constraint. For this reason the monotone constraints have always been considered *"hard to push"*, until the recent proposal of ExAnte [5]. In that work we have shown how monotone constraints can be exploited together with the frequency constraints by means of data-reduction. A transaction which does not satisfy a monotone constraint (recall here that a transaction is an itemset) can be deleted by the transaction database. This way, pushing monotone constraints does not reduce antimonotone pruning opportunities, on the contrary, such opportunities are boosted. Dually, pushing antimonotone constraints boosts monotone pruning opportunities: the two components strengthen each other recursively. This idea has been generalized in an Apriori-like computation in ExAMiner [4].

A succinct constraint $\mathcal{C}_S$ is such that, whether an itemset $X$ satisfies it or not, can be determined based on the singleton items which are in $X$. Informally, given $A_1$, the set of singleton items satisfying a succinct constraint $\mathcal{C}_S$, then any set $X$ satisfying $\mathcal{C}_S$ is based on $A_1$ , i.e. $X$ contains a subset belonging to $A_1$ (for the formal definition of succinct constraints see [11]). A $\mathcal{C}_S$ constraint is *pre-counting pushable*, i.e. it can be satisfied at candidate-generation time: these constraints are pushed in the level-wise computation by substituting the usual *generate_apriori* procedure, with the proper (w.r.t. $\mathcal{C}_S$) candidate generation procedure.

In Section 5 we describe how these properties of constraints are exploited by the optimized operational semantics of our framework.

## 3 Frequent Pattern Queries

In this Section going through a rigorous identification of all its basic components we provide a definition of frequent pattern query, i.e. a query defining a frequent pattern mining task over a relational database $\mathcal{D}$.

**Definition 6 (Mining View).** Given a database $\mathcal{D}$ a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$, explicitly indicated in the frequent pattern query as data source, is named *mining view.*

**Definition 7 (Transaction id).** Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Any subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ can be used as *transaction id.*

**Definition 8 (Circumstance attribute).** Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, we define any attribute $A \in sch(R)$ where $R$ is a relation in $preds(D)$ *circumstance attribute* provided that $A \notin \mathcal{T}$ and the functional dependency $\mathcal{T} \rightarrow A$ holds for in $\mathcal{D}$.

**Definition 9 (Item attribute).** Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, let $Y = \{y | y \in sch(\mathcal{V}) \backslash \mathcal{T} \wedge \mathcal{T} \rightarrow y \textit{ does not hold}\}$; we define an attribute $\mathcal{A} \in Y$ an *item attribute* provided the functional dependency $\mathcal{T}\mathcal{A} \rightarrow Y \backslash \mathcal{A}$ holds in $\mathcal{D}$.

**Proposition 10.** *Given a relational database $\mathcal{D}$, a triple $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$ denoting the mining view $\mathcal{V}$, the transaction id $\mathcal{T}$, the item attribute $\mathcal{I}$, uniquely identifies a transactional database, as defined in Definition 2.*

**Definition 11 (Descriptive attribute).** Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, and given $\mathcal{A}$ as item attribute; we define *descriptive attribute* any attribute $X \in sch(R)$ where $R$ is a relation in $preds(\mathcal{D})$, provided the functional dependency $\mathcal{T}\mathcal{A} \rightarrow X$ holds in $\mathcal{D}$.

Consider the mining view: `sales(tID, locationID, time, product, quantity)` where each attribute has the intended semantics of its name and with `tID` acting as the transaction id. Since the functional dependency {`tID`} $\rightarrow$ {`locationID`} holds, `locationID` is a circumstance attribute. The same is true for `time`. We also have {`tID, product`} $\rightarrow$ {`quantity`}, thus `product` is an item attribute, while `quantity` is a descriptive attribute.

Note that, from the previous definitions, *transaction id* and the *item attribute* must be part of the mining view, while circumstance and descriptive attributes could be also in other relations. Constraints, as introduced in the previous section, are defined on item attributes and descriptive attributes. Constraints over the *transaction id* or over circumstance attributes are not real constraints since they can be seen as selection conditions on the transactions to be mined and thus they can be satisfied in the definition of the *mining view*. Next definition lists all kinds of constraint that we admit in our framework.

| **Constraint** | **Description** |
|---|---|
| $s \supseteq \{a_1, \ldots, a_n\}$ | itemset contains |
| $s \subseteq \{a_1, \ldots, a_n\}$ | itemset domain |
| $\|s\| \, \theta \, m$ | cardinality constraint |
| $s.d \supseteq S$ | descriptive attribute contains |
| $s.d \subseteq S$ | descriptive attribute domain |
| $aggr\{d \mid i \in s \wedge P(i, \ldots, d)\} \, \theta \, m$ | aggregate on descriptive attribute |

**Fig. 3.** Bound constraints for frequent pattern query.

**Definition 12 (Bound constraints).** Given a database $\mathcal{D}$, a mining view $V(t, i, \ldots)$ where $t$ indicates the *transaction id* and $i$ denotes the *item attribute*, a relation $P(i, \ldots, d) \in preds(\mathcal{D})$ where $d$ indicates a descriptive attribute, all kinds of constraint admitted in a frequent pattern query are listed in Figure 3. The following notation is adopted:

- $s$ is an itemset;
- $a_1, \ldots, a_n$ are items;
- $d_1, \ldots, d_n$ are numeric constants of a descriptive attribute;
- $m$ is a numeric constant;
- $S$ is a set-valued or discrete constant;
- $\theta \in \{\leq, \geq, =\}$
- $aggr \in \{min, max, sum, avg, count, range\}$

**Definition 13 (Frequent pattern query).** Given a database $\mathcal{D}$, a frequent pattern query is a quintuple $\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle$ denoting the mining view $\mathcal{V}$, the transaction id $\mathcal{T}$, the item attribute $\mathcal{I}$, the minimum support threshold $\sigma$, and a conjunction of bound constraints $\mathcal{C}$.

The result of a frequent pattern query is a binary relation recording the set of itemset which satisfy $\mathcal{C}$ and are frequent in the transaction database $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$ w.r.t. $\sigma$ and their supports:

$$freq_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle}(I, S) \equiv \{(I, S) \mid \mathcal{C}(I) \ \wedge \ supp_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle}(I) = S \ \wedge \ S \geq \sigma\}$$

*Example 14.* A frequent pattern query for the `sales` table in Figure 2 (a), and the `product` table in Figure 2 (c), querying itemsets having a support $\geq 3$ (transactions are made grouping by customer and date), and having a total price $\geq 30$, could be simply defined as:

$$freq_{\langle sales, \{date, cust\}, item, 3, sum\{p \mid i \in I \wedge product(i,p,t)\} \geq 30 \rangle}(I, S) \equiv$$

$$\{(I, S) \mid sum\{p \mid i \in I \wedge product(i, p, t)\} \geq 30 \wedge supp_{\langle sales, \{date, cust\}, item \rangle}(I) = S \wedge S \geq 3\}$$

The result of such query is a relation `(I,S)` with the following two entries: `({beer, wine},4)` and `({beer, wine, chips},3)`.

## 4 Inductive Rules for Frequent Patterns

In this Section we provide the syntactic sugar to express the frequent pattern queries defined above. As already stated, in our language we drop the *user-defined aggregates* approach of $\mathcal{LDL}$-$\mathcal{M}$ine, and we define an inductive rule simply as a conjunction of sentences about the desired patterns (a conjunction of constraints).

In the following we provide a brief overview of the terms of our language to express frequent pattern queries. For the full syntax of our language, as well as for the definition of safe rules, see [2].

**Definition 15.** An inductive rule is a rule $H \leftarrow B_1, \ldots B_n$ such that:

- $B_1, \ldots B_n$ is a conjunction of inductive sentences, possibly containing deductive predicates.

– $H$ is the predicate representing the induced pattern.

Since we are modelling frequent pattern queries, a frequency inductive sentence will always be present in the body of an inductive rule.

**Definition 16 (Frequency inductive sentence).** Let $P$ be a variable for the induced frequent pattern, and $TDB$ a transaction database (a set of sets), then $freq(P, TDB)$ is the frequency inductive sentence which computes $supp_{TDB}(P)$.

Since we are dealing with frequent itemsets, sets will be first class citizens on which inductive sentences will be defined. For the same reason, traditional set operations ($\subseteq, \in, \notin$) will be useful to write inductive rules. A set can be a set of numbers, a set of strings, a set of variables or a set of sets. Moreover, since we want to compute patterns from transaction databases which can be in relational form, a set of sets can be built by grouping values of an attribute, as in the following Definition.

**Definition 17 (Pseudo-aggregation sentence).** Given a relation $R$ such that $X \in sch(R)$ and $Y \subset sch(R)$. The sentence $\langle X|Y \rangle$ denotes the pseudo aggregation on attribute $X$ grouping by attributes in $Y$.

In other words, if the term $\langle \mathtt{I}|\{\mathtt{D}, \mathtt{C}\} \rangle$ appears in an inductive rule containing the predicate `sales(D,C,I,Q)`; it corresponds to the pseudo-aggregate defined by the Datalog++ deductive rule:

$$\mathtt{sales}(\mathtt{D}, \mathtt{C}, \langle \mathtt{I} \rangle) \leftarrow \mathtt{sales}(\mathtt{D}, \mathtt{C}, \mathtt{I}, \mathtt{Q}).$$

which transforms the table in Figure 2(a) in the table in Figure 2(b).

Finally, in our inductive query language we need to express aggregates on descriptive attributes, in order to write constraints as those ones introduced in the previous section. The following example clarifies the relationship between the inductive and the deductive part of an inductive database, as well as the difference between the aggregate-based approach and our framework.

*Example 18.* Consider the frequent pattern query in Example 14: find itemsets having a support $\geq 3$ (transactions are made grouping by customer and date), and having a total price $\geq 30$.

In our framework we have two choices:

1. write an inductive rule to compute all frequent itemsets having a support $\geq 3$ and a deductive to selects those frequent itemsets which satisfy the constraint of having a total price $\geq 30$;
2. write the constraint directly in the body of the inductive rule defining the interesting patterns.

By a semantical point of view the two approaches are equivalent: they give the same result. By a purely computational point of view the first choice corresponds to a *generate-and-test* approach, while the second choice corresponds to a *constraint pushing* approach which is much more efficient.

In the aggregate-based approach of $\mathcal{LDL}$-$\mathcal{M}$ineonly the first choice is possible. We need a pseudo-aggregate rule to de-normalize the data source, the inductive rule based on the user defined `patterns` aggregate, a rule based on the sum aggregate to compute total sum of prices, and finally a rule to select the resulting patterns.

$$\begin{array}{ll}
\mathtt{sales(D,C,\langle I\rangle)} & \leftarrow \mathtt{sales(D,C,I,Q).} \\
\mathtt{frequentPatterns(patterns\langle(3,S)\rangle)} & \leftarrow \mathtt{sales(D,C,S).} \\
\mathtt{sumFP(S,N,sum\langle P\rangle)} & \leftarrow \mathtt{frequentPatterns(S,N),item(L,T,P),member(L,S).} \\
\mathtt{answer(S,N)} & \leftarrow \mathtt{sumFP(S,N,SP),SP >= 30.}
\end{array}$$

In our language, we can express that query with the following inductive rule:

$$\begin{array}{ll}
\mathtt{frequent_pattern(S,N)} \leftarrow & \mathtt{N = freq(S,X),\ X = \langle I|\{D,C\}\rangle,\ sales(D,C,I,Q),} \\
& \mathtt{N >= 3,\ L \in S,\ sum(P,product(L,P,T)) >= 30.}
\end{array}$$

Observe how in this rule all the basic components of a frequent pattern query are expressed. First of all we have the *mining view* declaratively indicated as a predicate in the body of the rule. Then we have the *transaction id* expressed in the right part of the pseudo-aggregation term, and the *item attribute* in the left part. Moreover we have two variables to indicate computed patterns and their support, which is constrained to be no less than a given *min-sup* threshold.

One could wish to return as result of a frequent pattern query additional information regarding the induced pattern. Therefore we allow inductive rules with more than two variables in the head. For sake of consistency, the additional variables must be aggregated at the itemset level. In other words, only variables describing aggregation of descriptive attributes are allowed.

For instance, we can ask to return together with itemsets and their supports, also the total sum of prices of an itemset:

$$\begin{array}{ll}
\mathtt{frequent_pattern(S,N,TP)} \leftarrow & \mathtt{N = freq(S,X),\ X = \langle I|\{D,C\}\rangle,\ sales(D,C,I,Q),} \\
& \mathtt{N >= 3,\ L \in S,\ TP = sum(P,product(L,P,T)),\ TP >= 30.}
\end{array}$$

As suggested by Example 18, each inductive rule is equivalent to one or more Datalog++ deductive rules. This observation is used to provide a declarative formal semantics for our language. In [2] formal semantics of the new inductive rules is provided by showing that exists a unique mapping from each safe inductive rule of the language to a Datalog++ program with aggregates (we shall omit details for lack of space). Thanks to this mapping we can define the formal declarative semantics of an inductive rule as the *iterated stable model* of the corresponding Datalog++ program [7].

The next example shows how to define an association rules mining tasks in our framework. More complex examples of frequent pattern queries can be found in [2].

*Example 19 (Association Rules).* It is well known that the task of computing association rules is performed by dividing it in two subproblems. In the first one, which is the real mining task, itemsets which are frequent for the given minimum support threshold are computed. In the second subproblem, which can be seen as a post-processing filtering task, the available association rules, for the given confidence threshold, are extracted from the frequent itemsets computed during the mining phase. In our framework we compute frequent itemsets by means of an inductive rule, and association rules by means of a deductive Datalog++ rule.

Suppose we want to compute simple association rules from the table in Figure 2(a), grouping transactions by day and customer, and having support more than 4 and confidence more than 0.6. The first rule is an inductive rule which defines the computation of patterns with an absolute support greater than 4.

$$\begin{aligned}
\texttt{frequentPatterns}(\texttt{Set},\texttt{Supp}) &\leftarrow \texttt{Supp} = \texttt{freq}(\texttt{Set},\texttt{X}),\ \texttt{sales}(\texttt{D},\texttt{C},\texttt{I},\texttt{Q}),\\
&\quad\ \texttt{X} = \langle \texttt{I} | \{\texttt{D},\texttt{C}\}\rangle,\ \texttt{Supp} >= 4.\\
\texttt{rules}(\texttt{Left},\texttt{Right},\texttt{Supp},\texttt{Conf}) &\leftarrow \texttt{frequentPatterns}(\texttt{A},\texttt{Supp}),\texttt{frequentPatterns}(\texttt{R},\texttt{S}_1),\\
&\quad\ \texttt{subset}(\texttt{R},\texttt{A}),\texttt{difference}(\texttt{A},\texttt{R},\texttt{L}),\texttt{Conf} = \texttt{Supp/S1},\texttt{Conf} >= 0.6.
\end{aligned}$$

The second rule is a Datalog++ deductive rule which computes the available association rules from the frequent itemsets (the result of the evaluation of the first rule is in Figure 4(a), while the result of the evaluation of the second rule is in Figure 4(b)).

## 5 Constraint Pushing Optimization

In Section 2 we have briefly reviewed constraints in frequent pattern mining and their properties. Such properties can be exploited for pruning the search space, and hence obtaining an optimized evaluation of frequent pattern queries. Following this approach, we systematically analyze each kind of constraint admitted in our language, and based on its properties, we define the constraint manager operator $\mathcal{CM}$ for the constraint pushing in the computation.

Similarly to the work done in [11], we divide all other kind of constraints in classes according to their properties. The main difference here is that, for us, monotone constraints are no longer hard constraints, since we have developed efficient techniques to push them in the computation, as those ones described in [3–5]. Therefore, we distinguish between 5 classes of constraints:

$\mathcal{C}_{AM}$ **constraints that are antimonotone but not succinct:** antimonotone constraints are exploited as usual to prune the search space in the level-wise, Apriori-like computation. They are checked together with frequency as a unique antimonotone constraint. Moreover, they are used in a data reduction fashion too, exploiting the real amalgam of antimonotonicity and monotonicity as described in [4, 5].

| Set | Supp |
|---|---|
| {beer} | 6 |
| {chips} | 7 |
| {wine} | 5 |
| {pasta} | 5 |
| {beer,chips} | 4 |
| {beer,wine} | 4 |

(a)

| Left | Right | Supp | Conf |
|---|---|---|---|
| {beer} | {chips} | 4 | 0.66 |
| {beer} | {wine} | 4 | 0.66 |
| {wine} | {beer} | 4 | 0.82 |

(b)

**Fig. 4.** (a) The `frequentPatterns` table and (b) the `rules` table which are the result of the evaluation of rules in Example 19.

$\mathcal{C}_M$ **constraints that are monotone but not succinct:** Monotone constraints, are exploited to reduce the input data and to prune the search space as described in [4, 5].

$\mathcal{C}_{AMS}$ **constraints that are both antimonotone and succinct:** constraints which are both antimonotone and succinct, can be satisfied before any mining takes place, by reducing the set of candidates 1-itemsets, to those items which satisfy such constraints as described in [11]. However, we can exploit them also in a data reduction fashion [4, 5].

$\mathcal{C}_{MS}$ **constraints that are both monotone and succinct:** are exploited as monotone constraints, by using them to reduce the data and the search space, and as succinct constraint in the candidate generation procedure, in the style of [11].

$\mathcal{C}_H$ **hard constraints:** Hard constraints, i.e. constraints which are neither antimonotone, nor monotone, nor succinct, are used inducing weaker constraints which exhibits some nice properties that allows pushing in the computation, and then they are checked at the end of the computation. In particular in our small language, as hard constraints, we have only the family of constraints based on the `avg` aggregate. The constraint $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ induces the weaker constraint $min\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$, which is both monotone and succinct and therefore it will be pushed in the computation to reduce the data and the search space. Analogously the constraint $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ induces the weaker, succinct and monotone, constraint $max\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$. However at the end of the computation, frequent itemsets which satisfy all constraints will be checked against the `avg` based constraint.

A particular kind of constraint is the cardinality constraint: when it is in the form $|s| \geq m$, it is a monotone constraint, and as that is treated. When it is in the form $|s| \leq m$, is an antimonotone constraint, but it differs from the other antimonotone constraints since it does not need to be checked for all itemsets, since the computation itself is level-wise (at iteration $k$ we count frequency of $k$-itemsets). Thus this constraint just imposes a stopping condition (level $m$) to the level-wise computation. We create a special class for this constraint: $\mathcal{C}_{Stop}$.

Following the above consideration we define a new operator, that puts each constraint in the query in the proper class. All these classes, once populated, will feed the mining algorithm as parameters.

**Definition 20 (Constraints manager).** Given a conjunction of constraints $\mathcal{C}$, we define the constraints manager operator, $\mathcal{CM}(\mathcal{C})$ as the operator which takes all constraints in $\mathcal{C}$, and put them in one or more class of constraints, as described in Figure 5.

Once the rule parser operator and the constraint manager have prepared the computation, an optimized mining operator $\mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_M, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_H, \mathcal{C}_{Stop})$ will start the level-wise Apriori-like computation using all the constraints available in order to reduce as much as possible the input data and the search space. The details about how $\mathcal{M}$ is implemented, as well as the pseudo-code are reported in [2] and in Appendix A. The optimized operational semantics for our inductive rules is defined as in Figure 6.

| $\mathbf{C_{ID}}$ | **Constraint** $C$ | $\mathcal{CM}(C)$ |
|---|---|---|
| 1 | $s \supseteq \{a_1, \ldots, a_n\}$ | $\mathcal{C}_{AMS}$ |
| 2 | $s \subseteq \{a_1, \ldots, a_n\}$ | $\mathcal{C}_{MS}$ |
| 3 | $\lvert s \rvert \geq m$ | $\mathcal{C}_{M}$ |
| 4 | $\lvert s \rvert \leq m$ | $\mathcal{C}_{Stop}$ |
| 5 | $\lvert s \rvert = m$ | $\mathcal{C}_{M}$, $\mathcal{C}_{Stop}$ |
| 6 | $s.d \supseteq S$ | $\mathcal{C}_{MS}$ |
| 7 | $s.d \subseteq S$ | $\mathcal{C}_{AMS}$ |
| 8 | $min\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{MS}$ |
| 9 | $min\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{AMS}$ |
| 10 | $min\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{AMS}$, $\mathcal{C}_{MS}$ |
| 11 | $max\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{AMS}$ |
| 12 | $max\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{MS}$ |
| 13 | $max\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{AMS}$, $\mathcal{C}_{MS}$ |

| $\mathbf{C_{ID}}$ | **Constraint** $C$ | $\mathcal{CM}(C)$ |
|---|---|---|
| 14 | $sum\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{AM}$ |
| 15 | $sum\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{M}$ |
| 16 | $sum\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{AM}$, $\mathcal{C}_{M}$ |
| 17 | $count\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{AM}$ |
| 18 | $count\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{M}$ |
| 19 | $count\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{AM}$, $\mathcal{C}_{M}$ |
| 20 | $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{H}$, $\mapsto 8$ |
| 21 | $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{H}$, $\mapsto 12$ |
| 22 | $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{H}$, $\mapsto 8, 12$ |
| 23 | $range\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ | $\mathcal{C}_{AM}$ |
| 24 | $range\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ | $\mathcal{C}_{M}$ |
| 25 | $range\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$ | $\mathcal{C}_{AM}$, $\mathcal{C}_{M}$ |

**Fig. 5.** Constraints Manager Table.

Optimized operational semantics of an inductive rule $r$

1. **if** $\mathcal{SC}(r)$
2. **then**
3. $\quad \mathcal{RP}(r) = freq_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle}(t_1, \ldots, t_n)$;
4. $\quad \mathcal{Q}(\mathcal{V}) = MV$;
5. $\quad \mathcal{CM}(\mathcal{C}) \mapsto \mathcal{C}_{AM}, \mathcal{C}_{M}, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_{H}, \mathcal{C}_{Stop}$
6. $\quad \mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_{M}, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_{H}, \mathcal{C}_{Stop}) = S$;
7. **for all** $s \in S$ **return**$(s, t_1, \ldots, t_n)$;
8. **else return** *unsafe rule*

**Fig. 6.** Optimized operational semantics of an inductive rule $r$.

## 6 Conclusions

This paper has provided a formalization of frequent pattern queries over relational databases. This formalization has permitted to define an inductive language for frequent pattern queries, which is simple enough to be highly optimized and expressive enough to cover the most of interesting queries. Following the recente results in constrained frequent pattern mining algorithms, we have defined an optimized constraint-pushing operative semantics for the queries of the proposed language.

There are some issues left uncovered by this research, for which further investigations are needed. An important issue is how to tightly integrate the inductive engine (optimized algorithms for inductive query) with the deductive DBMS. This issue is strictly connected with many other open problems ranging from how to store and index frequent pattern query results to how to reuse them for incremental query refinement. We believe that all this class of problems must be attacked together with an unifying approach.

## References


1. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.
2. F. Bonchi. *Frequent Pattern Queries: Language and Optimizations.* PhD thesis, Computer Science Department, University of Pisa, Italy, December 2003.
3. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Adaptive Constraint Pushing in frequent pattern mining. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD03)*, 2003.
4. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. ExAMiner: Optimized level-wise frequent pattern mining with monotone constraints. In *Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03)*, 2003.
5. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Exante: Anticipated data reduction in constrained pattern mining. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD03)*, 2003.
6. F. Giannotti and G. Manco. Querying Inductive Databases via Logic-Based User-Defined Aggregates. In J. Rauch and J. Zitkov, editors, *Procs. of the European Conference on Principles and Practices of Knowledge Discovery in Databases*, number 1704 in Lecture Notes on Artificial Intelligence, pages 125–135, September 1999.
7. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Nondeterministic, Nonmonotonic Logic Databases. *IEEE Transactions on Knowledge and Data Engineering.* 32(2):165-189, October 2001.
8. B. Goethals. *Efficient Frequent Pattern Mining.* PhD thesis, transnational University of Limburg, Diepenbeek, Belgium, December 2002.
9. J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-based, multidimensional data mining. *Computer*, 32(8):46–50, 1999.
10. H. Mannila and H. Toivonen. Levelwise Search and Border of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 3:241–258, 1997.
11. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98).*


## A Appendix: The Optimized Mining Operator $\mathcal{M}$

In Figure 7 the pseudo-code for the mining operator $\mathcal{M}$ is reported. Knowledge of the algorithms Apriori [1], ExAnte [5], ExAMiner [4] and CAP [11] is required in order to understand the pseudo-code. The mining operator performs an ExAnte [5] preprocessing at level 1 (lines from 1 to 18), where:

- both $\mathcal{C}_M$ and $\mathcal{C}_{MS}$ are exploited in order to $\mu$-reduce the input database (lines 3 and 11);
- $\mathcal{C}_{AM}$ and $\mathcal{C}_{AMS}$ are exploited in order to $\alpha$-reduce but only at the first round (line 6): after this reduction they will not be checked again at level 1, since they can not shrink.

Then it starts an Apriori generate and test computation where:

- $\mathcal{C}_{Stop}$ is used as additional stopping condition (line 22);

- the **generate_apriori** [1] (lines 20 and 24) procedure exploits succinct monotone constraints, as defined in [11];
- the **count** procedure is substituted by an ExAMiner **count**&**reduce** (line 23) procedure as defined in [4]: it uses $\mathcal{C}_M$, $\mathcal{C}_{AM}$ and $\mathcal{C}_{MS}$ to reduce the dataset;
- the final test (line 27), check $L_i$ for satisfaction of $\mathcal{C}_M$ and $\mathcal{C}_H$.

Note that in the final test of constraints, we do not need to check $\mathcal{C}_{MS}$, since their satisfaction is assured by the candidate generation procedure. The mining operator $\mathcal{M}$ defined above, corresponds to an $\mathbf{ExAMiner_1}$ computation, since it loops only at the first level, and then goes on strictly level-wise using the **count**&**reduce** procedure typical of ExAMiner. However, it is enriched by pushing all possible constraints in the proper step of the computation.

Mining operator: $\mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_M, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_H, \mathcal{C}_{Stop})$

1. $Items := \emptyset$;
2. **forall** transactions $\langle \mathcal{T}, \{\mathcal{I}\}\rangle$ in $MV$ **do**
3.     **if** $\{\mathcal{I}\} \in Th(\mathcal{C}_M \cap \mathcal{C}_{MS})$
4.     **then forall** items $i$ in $\{\mathcal{I}\}$ **do**
5.         $i.count{+}{+}$;
6.         **if** $i.count == \sigma$ **and** $\{i\} \in Th(\mathcal{C}_{AM} \cap \mathcal{C}_{AMS})$
7.         **then** $Items := Items \cup i$;
8. $old\_number\_interesting\_items := |Items|$;
9. **while** $|Items| < old\_number\_interesting\_items$ **do**
10.     $MV := \alpha[MV]_{\mathcal{C}_{freq}}$;
11.     $MV := \mu[MV]_{\mathcal{C}_M \cap \mathcal{C}_{MS}}$;
12.     $old\_number\_interesting\_items := |Items|$;
13.     $Items := \emptyset$;
14.     **forall** transactions $\langle \mathcal{T}, \{\mathcal{I}\}\rangle$ in $MV$ **do**
15.         **forall** items $i$ in $\{\mathcal{I}\}$ **do**
16.             $i.count + +$;
17.             **if** $i.count == \sigma$ **then** $Items := Items \cup i$;
18. **end while**
19. $L_1 := Items;\ C_2 := generate\_apriori(L_1, \mathcal{C}_{MS});\ k := 2$;
20. **while** $C_k \neq \emptyset$ **and not** $\mathcal{C}_{Stop}$ **do**
21.     $L_k := count\&reduce(\mathcal{C}_{freq} \cap \mathcal{C}_{AM}, C_k, MV, \mathcal{C}_M \cap \mathcal{C}_{MS})$;
22.     $C_{k+1} := generate\_apriori(L_k, \mathcal{C}_{MS})$;
23.     k++;
24. **end while**
25. **return** $\bigcup_{i=1}^{k-1} L_i \cap Th(\mathcal{C}_M) \cap Th(\mathcal{C}_H)$

**Fig. 7.** Pseudo-code of the mining operator $\mathcal{M}$.