On Interactive Pattern Mining from Relational Databases

Francesco Bonchi[1], Fosca Giannotti[1], Claudio Lucchese[2,3], Salvatore Orlando[2,3], Raffaele Perego[3], and Roberto Trasarti[1]

[1] Pisa KDD Laboratory, ISTI - CNR,
Area della Ricerca di Pisa, Via Giuseppe Moruzzi 1, Pisa, Italy
[2] Computer Science Dep., University Ca' Foscari
Via Torino 155, Venezia Mestre, Italy
[3] Pisa HPC Laboratory, ISTI - CNR,
Area della Ricerca di Pisa, Via Giuseppe Moruzzi 1, Pisa, Italy

Abstract. In this paper we present ConQueSt, a constraint based querying system devised with the aim of supporting the intrinsically exploratory (i.e., human-guided, interactive, iterative) nature of pattern discovery. Following the *inductive database* vision, our framework provides users with an expressive constraint based query language which allows the discovery process to be effectively driven toward potentially interesting patterns. Such constraints are also exploited to reduce the cost of pattern mining computation. We implemented a comprehensive mining system that can access real world relational databases from which extract data. After a preprocessing step, mining queries are answered by an efficient pattern mining engine which entails several data and search space reduction techniques. Resulting patterns are then presented to the user, and possibly stored in the database. New user-defined constraints can be easily added to the system in order to target the particular application considered.

1 Introduction

According to the *inductive database* vision [16], the task of extracting useful and interesting knowledge from data is just an *exploratory* querying process, i.e., human-guided, iterative and interactive. The analyst, exploiting an expressive query language, drives the discovery process through a sequence of complex mining queries, extracts patterns satisfying some user-defined constraints, refines the queries, materializes the extracted patterns as first-class citizens in the database, combines the patterns to produce more complex knowledge, and cross-over the data and the patterns. Therefore, an Inductive Database system should provide the following features:

Coupling with a DBMS. The analyst must be able to retrieve the portion of interesting data (for instance, by means of SQL queries). Moreover, extracted patterns should also be stored in the DBMS in order to be further queried or mined (*closure principle*).

S. Džeroski and J. Struyf (Eds.): KDID 2006, LNCS 4747, pp. 42–62, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Expressiveness of the query language. The analyst must be able to interact with the pattern discovery system by specify declaratively how the desired patterns should look like and which conditions they should satisfy. The analyst is supposed to have a high-level vision of the pattern discovery system, without worrying about the details of the computational engine, in the same way as a database user has not to worry about query optimization. The task of composing all constraints and producing the most efficient mining strategy (execution plan) for a given query should be thus completely demanded to the underlying system.

Efficiency of the mining engine. Keeping query response time as small as possible is, on the one hand necessary, since our goal is to give frequent feedbacks to the user allowing realistic human-guided exploration. On the other hand, it is a very challenging task, due to the exponential complexity of pattern discovery computations. To this end, data and search space reduction properties of constraints should be effectively exploited by pushing them within the mining algorithms. Pattern discovery is usually a highly iterative task: a mining session is usually made up of a series of queries (exploration), where each new query adjusts, refines or combines the results of some previous queries. It is important that the mining engine adopts techniques for incremental mining; i.e. reusing results of previous queries, in order to give a faster response to the last query presented to the system, instead of performing again the mining from scratch.

Graphical user interface. The exploratory nature of pattern discovery imposes to the system not only to return frequent feedbacks to the user, but also to provide pattern visualization and navigation tools. These tools should help the user in visualizing the continuous feedbacks form the systems, allowing an easier and human-based identification of the fragments of interesting knowledge. Such tools should also play the role of graphical querying interface: the action of browsing pattern visualization should be tightly integrated (both by a conceptual and engineering point of view) with the action of iteratively querying.

Starting from the above requirements we designed ConQueSt, an exploratory pattern discovery system equipped with a simple, yet powerful, query language (named spql for *simple pattern query language*) and a user friendly interface for accessing the underlying DBMS. While designing ConQueSt query language, architecture and user interface, we have kept in mind all the tasks involved in the typical *knowledge discovery process* [11]: (*i*) source data selection, (*ii*) data preparation and pre-processing, (*iii*) pattern discovery and model building. The user supervises the whole process, not only defining the parameters of the three tasks, but also evaluating the quality of the outcome of each step and possibly re-tuning the parameters of any step. Moreover, the user is in charge of interpreting and evaluating the extracted knowledge, even if the system must provide adequate support for this task. As we will show in this paper, the three main tasks of the knowledge discovery process are represented both in the query language and in the architecture of the ConQueSt system.

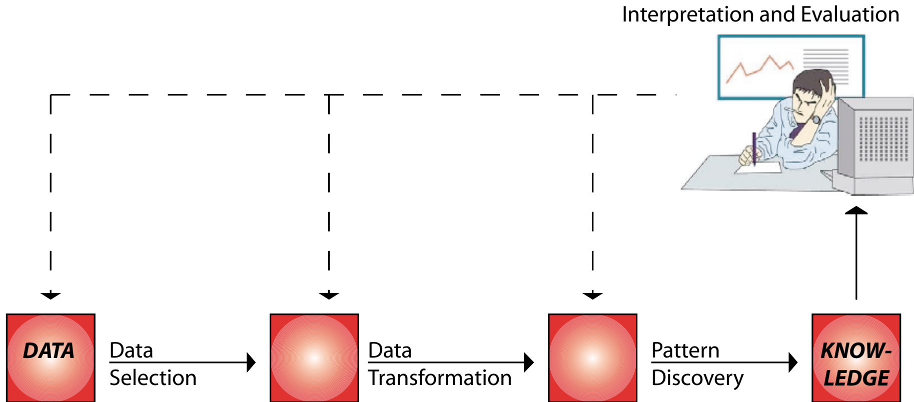


Fig. 1. ConQueSt knowledge discovery process

ConQueSt is the result of a joint work of the Pisa KDD (*Knowledge Discovery and Delivery*) and HPC (*High Performance Computing*) Laboratories: it is built around a scalable and high-performance constraint-based mining engine exploiting state-of-the-art constraint pushing techniques, as those ones developed in the last three years by our two labs [24,6,5,7,8].

1.1 Constraint Based Pattern Mining

Devising fast and scalable algorithms, able to crunch huge amount of data, has been so far one of the main goal of data mining research. Recently, researchers realized that in many practical cases it does not matter how much efficiently knowledge is extracted, since the volume of the results themselves is often embarrassing, and creates a second order mining problem for the human expert. This situation is very common in the case of association rules and frequent pattern mining [2], where the identification of the fragments of interesting knowledge, blurred within a huge quantity of mostly useless patterns, is very difficult.

The constraint-based pattern mining paradigm has been recognized as one of the fundamental techniques for inductive databases: user-defined constraints drive the mining process towards potentially interesting patterns only. Moreover, they can be pushed deep inside the mining algorithm in order to deal with the exponential search space curse, achieving better performance [28,23,15]. Constraint-based frequent pattern mining has been studied a lot as a query optimization problem, i.e., developing efficient, sound and complete evaluation strategies for constraint-based mining queries. To this aim, properties of constraints have been studied comprehensively, e.g., *anti-monotonicity*, *succinctness* [23,19], *monotonicity* [18,9,5], *convertibility* [25], *loose anti-monotonicity* [8], and on the basis of such properties efficient computational strategies have been defined. The formal problem statement follows.

Definition 1 (Constrained Frequent Itemset Mining). *Let* $\mathcal{I} = \{x_1, ..., x_n\}$ *be a set of distinct 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* $\mathcal{I}$*. A transaction database* $\mathcal{D}$ *is a bag of itemsets* $t \in 2^{\mathcal{I}}$*, usually called transactions. 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:* $Th(\mathcal{C}) = \{X \in 2^{\mathcal{I}} \mid \mathcal{C}(X)\}$*. The support of an itemset* X *in database* $\mathcal{D}$*, denoted* $supp_{\mathcal{D}}(X)$*, is the number of transactions which are superset of* X*. Given a user-defined minimum support, denoted* σ*, an itemset* X *is called frequent in* $\mathcal{D}$ *if* $supp_{\mathcal{D}}(X) \geq \sigma$*. This defines the minimum frequency constraint:* $\mathcal{C}_{freq[\mathcal{D},\sigma]}(X) \Leftrightarrow supp_{\mathcal{D}}(X) \geq \sigma$*. In general, given a conjunction of constraints* $\mathcal{C}$ *the constrained frequent itemsets mining problem requires to compute* $Th(\mathcal{C}_{freq}) \cap Th(\mathcal{C})$*.*

While developing ConQueSt we have tried to reduce as much as possible the gap existing between real-world data stored in relational DBMS, and the constraint-based pattern discovery paradigm, as defined above. In fact, the data is usually stored in relational databases, and thus in the form of relations and not of transactions. In sections 2 and 3.2 we explain how transactions are defined and constructed both at the query language and at the system level. Moreover, the constraint-based mining paradigm assumes that the constraints are defined on attributes of the items that are in functional dependency with the items. This rarely the case in real-world data: just think about the price of one item in a market over a period of one month. ConQueSt provides support to solve this cases, allowing the user to reconstruct the ideal situation of functional dependency, for instance, by choosing to take the average of prices of the item in the period as price attribute.

In this paper we provide an overview of ConQueSt's main design choices and features. The paper is organized as follows. In Section 2 we provide a brief overview of the SPQL language which is at the basis of our system. Then in Section 3 we provide an high level description of ConQueSt's architecture, we then describe the three main modules in the sections that follow. In particular, Section 3.1 describes the graphical user interface and how the interactions between the user and the system actually happens; Section 3.2 describes the query interpreter and pre-processor modules; Section 3.3 describes the mining engines and the algorithmic choices underlying it. Finally in Section 4 we discuss other existing mining systems and query languages, and in Section 5 we draw some conclusions.

2 A Simple Data Mining Query Language

According to the constraint-based mining paradigm, the data analyst must have a high-level vision of the pattern discovery 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 pattern discovery system, using a query language. The analyst just needs to declaratively specify

in the pattern discovery query how the desired patterns should look like and which conditions they should obey (a set of constraints). Such rigorous interaction between the analyst and the pattern discovery system, can be implemented following [20], 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 w.r.t. the data. 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.

Definition 2. *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*

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

The selection predicate q indicates whether a pattern s is considered interesting. In the constraint-based paradigm, such selection predicate q is defined by a conjunction of constraints. In this Section, going through a rigorous identification of all its basic components, we provide a definition of constraint-based frequent pattern mining query over a relational database DB [4].

The first needed component is the data source: which table must be mined for frequent patterns, and which attributed do identify transactions and items.

Definition 3 (Data Source). *Given a database* DB *any relational expression* $\mathcal{V}$ *on* $preds$(DB) *can be selected as* data source*.*

Definition 4 (Transaction id). *Given a database* DB *and a relation* $\mathcal{V}$ *derived from* $preds$(DB)*. Let* $\mathcal{V}$ *with attributes* $sch(\mathcal{V})$ *be our data source. Any subset of attributes* $\mathcal{T} \subset sch(\mathcal{V})$ *can be selected as transaction identifier, and named* transaction id*.*

Definition 5 (Item attribute). *Given a database* DB *and a relation* $\mathcal{V}$ *derived from* $preds$(DB)*. Let* $\mathcal{V}$ *with attributes* $sch(\mathcal{V})$ *be our data source. Given a subset of attributes* $\mathcal{T} \subset sch(\mathcal{V})$ *as transaction id, let* $Y = \{y | y \in sch(\mathcal{V}) \backslash \mathcal{T} \wedge \mathcal{T} \rightarrow y$ does not hold$\}$*; we define an attribute* $\mathcal{I} \in Y$ *an* item attribute *provided the functional dependency* $\mathcal{T}\mathcal{I} \rightarrow Y \backslash \mathcal{I}$ *holds in* DB*.*

Proposition 1. *Given a relational database* DB*, a triple* $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$ *denoting the data source* $\mathcal{V}$*, the transaction id* $\mathcal{T}$*, the item attribute* $\mathcal{I}$*, uniquely identifies a transactional database, as defined in Definition 1.*

We next distinguish between attributes which describe items (descriptive attributes), from attribute which describe transactions (circumstance attributes).

Definition 6 (Circumstance attribute). *[12] Given a database* DB *and a relation* $\mathcal{V}$ *derived from* $preds$(DB)*. Let* $\mathcal{V}$ *with attributes* $sch(\mathcal{V})$ *be our data source. Given a subset of attributes* $\mathcal{T} \subset sch(\mathcal{V})$ *as transaction id, we define any attribute* $\mathcal{A} \in sch(R)$ *where* R *is a relation in* $preds$(DB) circumstance attribute *provided that* $\mathcal{A} \notin \mathcal{T}$ *and the functional dependency* $\mathcal{T} \rightarrow \mathcal{A}$ *holds in* DB*.*

Definition 7 (Descriptive attribute). *Given a database* DB *and a relation* $\mathcal{V}$ *derived from* $preds$(DB). *Let* $\mathcal{V}$ *with attributes* $sch(\mathcal{V})$ *be our data source. Given a subset of attributes* $\mathcal{T} \subset sch(\mathcal{V})$ *as transaction id, and given* $\mathcal{I}$ *as item attribute; we define* descriptive attribute *any attribute* $\mathcal{A} \in sch(R)$ *where* R *is a relation in* $preds$(DB), *provided the functional dependency* $\mathcal{I} \rightarrow \mathcal{A}$ *holds in* DB.

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

Constraints, as introduced in the previous Section (see Definition 1), describes properties of itemsets, i.e., a constraint $\mathcal{C}$ is a boolean function over the domain of itemsets: $\mathcal{C} : 2^{\mathcal{I}} \rightarrow \{true, false\}$. According to this view, constraints are only those ones defined on item attributes (Definition 5) or descriptive attributes (Definition 7).

Constraints defined over the *transaction id* (Definition 4) or over circumstance attributes (Definition 6) are not constraints in the strict sense. Indeed, 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*. Consider the relation: `sales(tID, locationID, time, product, price)` where each attribute has the intended semantics of its name and with `tID` acting as the transaction id. Since the functional dependency {`tID`} → {`locationID`} holds, `locationID` is a circumstance attribute. The constraints `locationID` ∈ {`Florence, Milan, Rome`} is not a real constraint of the frequent pattern extraction, indeed it is a condition in the mining view definition, i.e., it is satisfied by imposing such condition in the relational expression defining the mining view (for a deeper insight on circumstance attributes and constraints defined over them see [27,12]).

We have provided all the needed components for defining a constraint-based frequent pattern query as follows.

Definition 8 (Constraint-based frequent pattern query). *Given a database* DB, *let the quintuple* $\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle$ *denotes the mining view* $\mathcal{V}$, *the transaction id* $\mathcal{T}$, *the item attribute* $\mathcal{I}$, *the minimum support threshold* σ, *and a conjunction of constraints on itemsets* $\mathcal{C}$.

The primitive for constraint-based itemset mining takes in input such quintuple and returns a binary relation recording the set of itemsets which satisfy $\mathcal{C}$ *and are frequent (w.r.t.* σ*) in the transaction database* $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$, *and their supports:*

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

The SPQL query language that is at the basis of ConQueSt, is essentially syntactic sugar, in SQL-like style, to express constraint-based frequent pattern queries like $freq(\mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C})$. It is a superset of SQL, in a double sense: first any SPQL query contains a SQL query needed to define the data source; second, in ConQueSt we allow the user to define any SQL query, which could be useful, for instance, to pre-process the data or post-process the extracted patterns.

Table 1. An example SPQL mining query

```
1. MINE PATTERNS WITH SUPP>= 5 IN
2. SELECT product.product_name, product.gross_weight, sales_fact_1998.time_id,
   sales_fact_1998.customer_id, sales_fact_1998.store_id
3. FROM [product], [sales_fact_1998]
4. WHERE sales_fact_1998.product_id=product.product_id
5. TRANSACTION sales_fact_1998.time_id, sales_fact_1998.customer_id,
   sales_fact_1998.store_id
6. ITEM product.product_name
7. ATTRIBUTE product.gross_weight
8. CONSTRAINED BY Sum(product.gross_weight)<=30
```

In Table 1 we report a true SPQL mining query, defined within ConQueSt on the famous `foodmart2000` datamart. A simple SPQL query consists of four parts:

1. the user-defined minimum support threshold σ in line 1;
2. the SQL style `SELECT` statement to specify the data source $\mathcal{V}$ to be extracted from the DB (lines 2–4);
3. the mining view definition by means of `TRANSACTION` (to identify $\mathcal{T}$), `ITEM` (to identify $\mathcal{I}$), and of `ATTRIBUTE` on which constraints are defined (lines 5–7);
4. the conjunction of constraints $\mathcal{C}$ that the extracted patterns must satisfy in line 8.

Since the data source is in relational form, a pre-processing step is needed to create a set of transactions, which are the input of any frequent pattern mining system. Transaction are created by grouping `ITEM` by the attributes specified in the `TRANSACTION` clause. For instance, in the query Table 1, transactions are built groping sales by time id, customer id and store id: this means that we consider a unique transaction when we got the same customer in the same store at the same time. It is worth noting that with this simple mechanism of defining transactions we can easily handle both the inter-attribute and the intra-attribute pattern mining cases. We have chosen a well defined set of classes of constraints. These constraints have been deeply studied and analyzed in the past few years, in order to find nice properties that can be used at mining time to reduce the computational cost. In particular the ConQueSt system is able to deal with anti-monotone, succinct [23], monotone [5], convertible [26] and loose anti-monotone [8] constraints. Such classes include all the constraints based on the aggregates listed in Table 2.

It is worth noting how all steps of the knowledge discovery process, i.e., (*i*) source data selection, (*ii*) data preparation and pre-processing and (*iii*) pattern discovery, are expressed within the typical SPQL query. In particular, the source data selection is expressed by means on the `select` statement, the preprocessing is expressed by means of items and transactions identification, and the mining is expressed by listing the desired constraints, including the frequency one.

Table 2. The set of available constraints

subset	subset	**supset**	superset
asubset	attributes are subset	**len**	length
asupset	attributes are superset	**acount**	attributes count
min	minimum	**max**	maximum
range	range	**sum**	sum
avg	average	**var**	variance
std	standard deviation	**spv**	sample variance
md	mean deviation	**med**	median

This is a SPQL query in its basilar form. Different kinds of SPQL query exist, since we have extended the language to accommodate the use of *soft constraints* [3], and *discretization tasks*. In ConQueSt we have introduced the possibility of defining queries according to the new paradigm of pattern discovery based on *soft constraints*[3], i.e., where constraints are no longer rigid boolean functions. This allows the user to describe what is the "shape" of the patterns of interest, and receive back those patterns that *"mostly"* exhibits such shape. The patterns are also sorted according to a measure of interestingness, i.e., how much a pattern agrees with the given description. Having this order, allows also to define top-k queries. In this paper we avoid entering in the details of these extensions. The interested reader can find an example of SPQL query using *soft constraints* in the paper [3] in this volume.

In Table 3 a portion of SPQL formal grammar definition is provided.

Table 3. A portion of SPQL formal grammar definition

```
<SpqlQuery> ::= (<SqlQuery>| <MineQuery> | <Discretize>)
<MineQuery> ::=<Header><br><SqlQuery><br><MiningDefinition><br><Constraints>
<Header> ::= MINE PATTERNS WITH SUPP >= <Number>
<MiningDefinition> ::= TRANSACTION <Transaction><br> ITEM
<Item><br>[ ATTRIBUTE <Attribute>]
<Transaction> ::= <Field>[<Separator><Transaction>]
<Item> ::= <Field>[<Separator><Item>]
<Attribute> ::= <Field>[<Separator><Attribute>]
<Field> ::= <String>.<String>
<Constraints> ::= CONSTRAINED BY <Function>
<Function> ::= (<FunctionM>(<Field>)<Op><Number> | <FunctionS>(<Field>)<Op><Set> |
               <FunctionN>()<Op><Number> ) [<Separator>(Function)]
<FunctionM> ::= (Minimum | Maximum | Range | Variance | Std_Deviation | Median | Average | Sum)
<FunctionS>::= (Subset_of | Superset_of | Attribute_Subset_of | Attribute_Superset_of )
<FunctionN>::= Length
<Op> ::= (>|<|>=|<=)
<Separator> ::= ,
<br> ::= \n
<Set> ::= <String>[<Separator><Set>]
<Discretize>::= DISCRETIZE <Field> AS <Field> <br> FROM <String> <br> IN
(<Method>| <Intervals>) BINS <br> SMOOTINGH BY <Smethod>
<Method> ::= (EQUALWIDTH | EQUALDEPTH)
<Smethod> ::= (AVERAGE | COUNT | BIN BOUNDARIES)
<Intervals>::= (<Number> <Separator> <Number>)[<Separator> <Intervals>]
<Number> ::= (0-9) [<Number>]
<String> ::= (a-z|A-Z|0-9) [<String>]
```

3 Architecture of the System

The ConQueSt architecture, as shown in Fig. 2, is composed of three main modules:

– the Graphical User Interface (GUI);
– the Query Interpreter and and Pre-processor (QIP);
– the Mining Engine (ME).

For portability reasons, the GUI and the QIP have been implemented in Java, while the ME was implemented in C++ in order to provide high performance during the most expensive task.

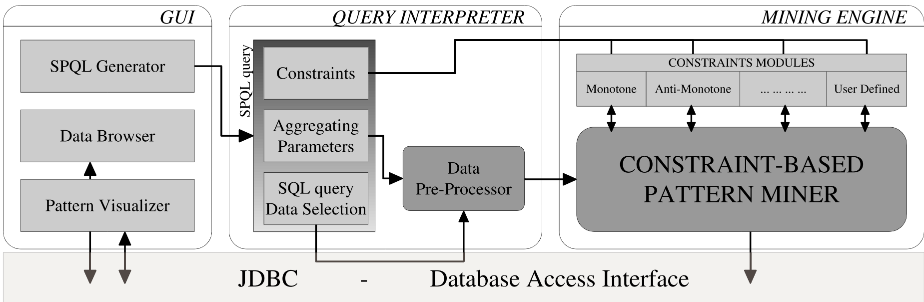


Fig. 2. ConQueSt architecture

In our vision, a mining system has to be tightly coupled with DMBS softwares, because databases are the place where data is. Our choice is to allow all the three main components to access independently a database. In fact, the GUI must show to the user the data present in the database, the QIP must retrieve the data of interested and prepare them for the mining engine, which will eventually store the discovered patterns in the database. To this end, the three components stand on a JDBC [1] abstraction layer, in order to provide independency from the particular database server software where data are stored. In fact, the JDBC API provides connectivity to a wide range of SQL databases. ConQueSt for instance, can retrieve data from PostgreSQL as well as from Microsoft Access database servers, and additional compatibility can be provided just by adding the JDBC plug-in provided by the database server software house.

The separation in these modules reflects the separation among different, well defined and independent tasks. In fact, every module could be a single software package running on a different machine. For instance, the GUI may run on the user machine, while the QIP may be located in a different site where a fast access to the database is provided, and finally the ME may be running on an high performance cluster serving many users with many different tasks. Finally, the JDBC layer allow us to ignore the physical location of the database server.

Actually, a communication protocol is established, flowing from the GUI, through the QIP and ending at the mining engine. The GUI, interactively with

the user, creates a SPQL query which is then sent to the query interpreter. The latter preprocesses the data of interest and translates part of the SPQL query in an list of constraints. These constraints, and a filesystem pointer to the preprocessed data are finally sent to the ME which can now start the mining process. As long as this simple protocol is fulfilled, every single component can increase its features and improve its functionalities independently from the others.

3.1 Graphical User Interface

The user interface (see a screen-shot in Figure 3) is designed not only to stand in between the user and the mining engine, but also to stand between the user and the data. Data is assumed to be in the form of a relational database. As soon as the user connects to the database, a set of information and statistics are collected and presented in many ways. The idea is to provide a simple and high level mean to the user to define the mining task. It is simple, since the user can reach his goal just by using user friendly mouse-clicks. Moreover, many high level information and statistics are provided. Finally, the GUI is complete, meaning that any operation related to the definition of a mining query can be done just by mouse-clicks without the need of editing an SPQL query by hand.However, in the case an expert user prefers to write a query by hand, or simply to change an existing one, ConQueSt's *inverse-parser* takes care of updating the GUI's modules on-the-fly, in such a way that exists always a perfect correspondence

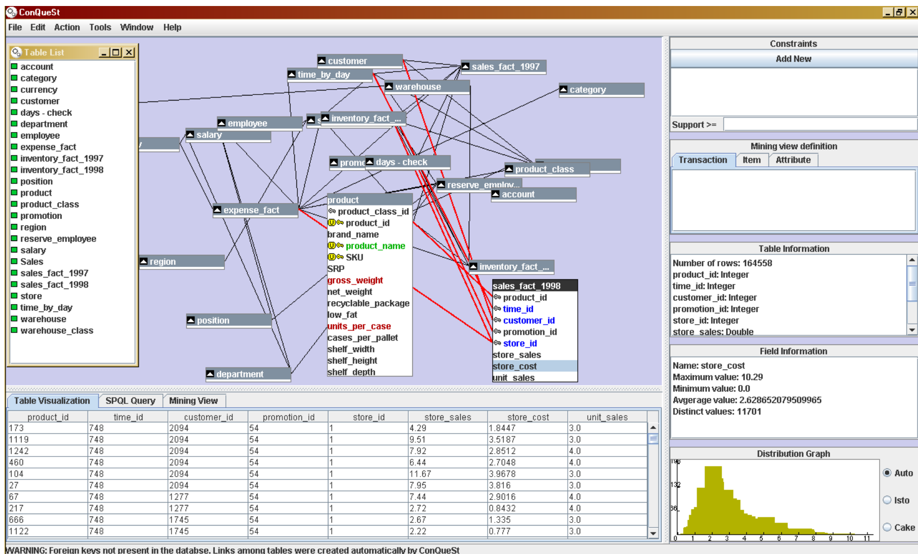
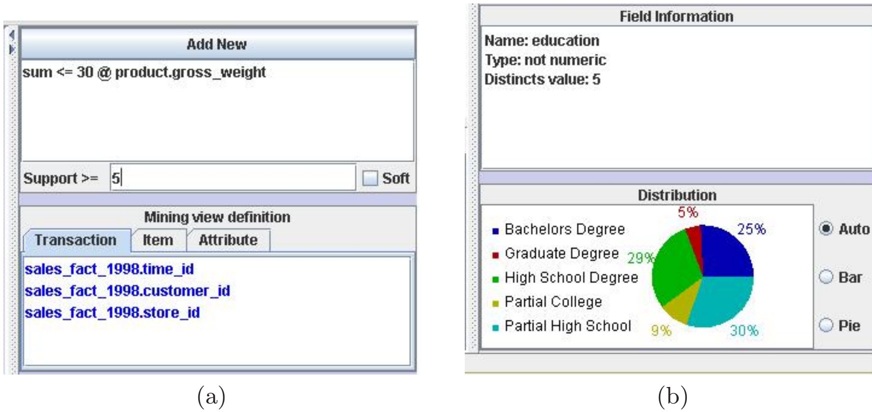


Fig. 3. ConQueSt graphical user interface



(a) (b)

Fig. 4. ConQueSt The upper-right corner of the GUI, corresponding to the query in Table 1 (a), and the lower-right corner with the description of the `education` attribute of the `customer` table (b)

between the SPQL query textually reported in the bottom panel of the GUI, and its definition contained in the central area and in the two upper-right corner panels of the GUI. In Figure 4(a), we report the two upper-right corner panels for the query in Table 1.

Navigating the Structure of the Database. Most of the GUI is dedicated to the *Desk Area*. Here the tables present in the database are showed in a shape of a graph structure. Each vertex of the graph represents a table, and the user may choose to see or not to see all the fields of the table. Each edge of the graph corresponds to a logical link between a foreign key and a primary key. Finally, a *Tables List* helps the user to select, to hide or to show any of the tables. This gives the user an high level view of the database, allowing him grasp all the relations and connections at a glance, and to focus on the portion of data of interest.

Table-Fields Information and Statistics. Every table maybe actually visualized in the *Table Visualization* panel, but aggregated information are more useful to the user. For this reason ConQueSt shows the data type of each field, statistics of the selected field (e.g. average, minimum, maximum) and a bar or pie chart of the distribution of the values of the selected field (see a screenshot in Figure 4(b)). This information may help the user in deciding the discretization parameters and the constraints thresholds.

Interactive SPQL query definition. The first part of the SPQL query consists in the mining view definition, i.e. the set of fields defining transactions, items and attributes. The user may set the mining view simply by right-clicking directly on the *Desk Area* the table-fields of interest. Those fields will be highlighted in the *Desk Area* and reported in the *Mining View Definition* panel. Whenever a mining view definition implicitly require relational joins

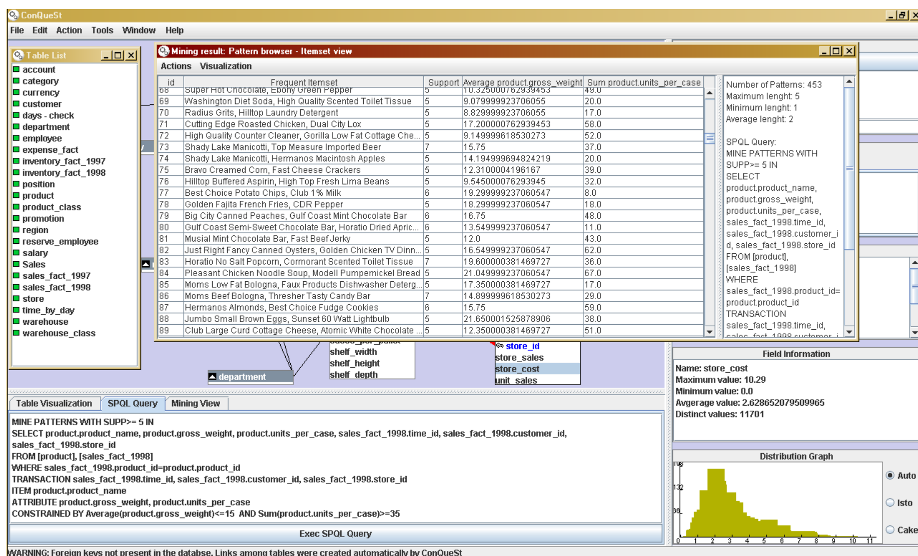


Fig. 5. The output of a mining query in the pattern browser

(e.g. transaction ids and item ids are in different table), they are automatically inserted in the final SPQL query. Constraints and respective threshold may also be set by right-clicking on the *Desk Area* or also using the *constrains* panel (in Figure 4). These facilities allows the user the fully define the mining task just by navigating the dataset graph and using mouse clicks.

Advanced SPQL query definition. At any moment, the user can edit by hand the query in the SPQL query panel. Any modification of the query will be reflected in the rest of the GUI, e.g. by updating the *Mining View Definition* panel. The possibility to edit directly the query, rather than using the GUI, does not provide any additional expressive power from a mining point of view. Anyway, since part of the SPQL query is pure SQL, we can allow the user to exploit complex SQL queries and additional constraints that are not part of the mining task, but rather they are part of the data preparation phase. Moreover, any SQL query can be submitted in place of an SPQL query, providing additional control to the analyst. Finally, before executing the mining task, a preview of the data in the transactional format, together with its items and attributes, is provided in the *Mining View* panel. This is helpful to check the output of the data preparation step before evaluating the query (or in other terms, before starting the mining).

Pattern Browser. Result of mining queries are shown to the user in a specialized interface, named *Pattern Browser* (see Figure 5. The Pattern Browser provides statistics on the query results, and various functionalities for interactive navigation the set of patterns, such as various kinds of visualizing the patterns, or sorting them. The pattern browser also shows the SPQL

query that generated the patterns, and allows the user to tune the query parameters according to his needs, for instance, strengthening or relaxing some constraints on-the-fly. This allows the user to quickly make the query converge towards the desired parameters, for instance, towards a reasonably sized solution set. In the pattern browser the user can also invoke some post-processing ore require to *materialize* the extracted patterns in the underlying database. At the moment the unique kind of post-processing implemented is the extraction of *association rules* from the patterns results set, but we plan to introduce more complex post-processing which uses extracted patterns as basic bricks to build global models as clustering or classifiers. Also the set of extracted association rules can be materialized as relational tables in the underlying database.

3.2 Query Interpreter and Pre-processor

The second module takes care of interpreting the SPQL query, retrieving from the underlying DBMS the data source, and preparing it for the mining phase.

The preprocessor receives from the GUI a well-formed SPQL query, and it is in charge to accomplish the data preparation step. In fact, the Mining Engine is not able to deal with a relational dataset, it can only read data in a proper format. This format is the one traditionally used in frequent itemsets mining contexts. The input dataset is a collection of transactions, where each transaction is a set of *items*. Each of these items is stored as an integer identifier. In the relational dataset an item may be a string, or even a floating point value, but to feed the mining engine these values have to be discretized or mapped.

Thus, the query interpreter, uses the mining view definition present in the SPQL query to retrieve from the original dataset the data of interest. These data are mapped and translated in a categorical format and finally stored on disk. The rest of the query, i.e. frequency and other constraints, are forwarded to the mining engine together with a pointer the the transactional dataset.

3.3 The Mining Engine

The last module constitutes the core of the system which mines the transactional dataset and extract the patterns. The mining core guarantees the scalability and performance of the system by exploiting efficient mining algorithms and data reduction techniques coming from the constraint-based mining paradigm.

The ConQueSt mining engine is based on DCI [24], a high performance, Apriori-like, frequent itemsets mining algorithm, with has the nice feature of being resource and data aware. It is resource aware, because, unlike many other algorithms, it performs the first iterations out-of-core, reducing the dataset and rewriting it directly to the disk. When the dataset is small enough, it is converted and stored as a vertical bitmap in main memory. It is data aware because its behavior changes in presence of sparse or dense datasets. It uses a compact representation in the case of sparse datasets, and detects highly correlated items to save bitwise works in the case of dense datasets. ConQueSt by inheriting the

same characteristics, is extremely robust and able to effectively mine different kinds of datasets, regardless of their size.

Although the ConQueSt mining engine adopts a level-wise Apriori-like visit of the lattice, thanks to its internal vertical bitwise representation of the dataset, and associated counting strategy, it performs better than other state-of-the-art algorithms that exploit a depth first visit. Moreover, as we have shown in our previous works [5,8], adopting a level-wise strategy has the great advantage of allowing the exploitation of different kinds of constraints all together by means of data-reduction. At each iteration of the mining process, the dataset is pruned by exploiting the independent data reductions properties of all user-specified constraints. Our framework, exploits a real synergy of all constraints that the user defines for the pattern extraction: each constraint does not only play its part in reducing the data, but this reduction in turns strengthens the pruning power of the other constraints. Moreover data-reduction induces a pruning of the search space, which in turn strengthens future data reductions. The orthogonality of the exploited constraint pushing techniques has a twofold benefit: first, all the techniques can be amalgamated together achieving a very efficient computation.

Moreover, since we have precisely identified classes of constraints which share nice properties, each class has been implemented as an independent module, which plays its role in precise points of the computation. Each module is then instantiated on the basis of the specific constraints supplied by the user. ConQueSt can be easily extended to cope with new user-defined constraints. In fact, it is not the constraint itself that performs data and search space reductions directly, but it is instead the overall framework which exploits constraints classes properties during the computation. Therefore, in order to define a novel constraint, and embed it in the computational framework, it is sufficient to communicate to the system to which classes (possibly more than one) it belongs.

4 Other Mining Query Languages

In this section we discuss other approaches to the data mining query language definition issue. For lack of space we focus only on the approaches most related to the ConQueSt proposal. We are aware that this presentation does not exhaustively cover the wide state-of-the-art of the research (and also the development) on data mining systems and query languages.

The problem of providing an effective interface between data sources and data mining tasks has been a primary concern in data mining. There are several perspectives upon which this interface is desirable, the most important ones being (*i*) to provide a standard formalization of the desired patterns and the constraints they should obey to; and (*ii*) to achieve a tighter integration between the data sources and the relational databases (which likely accommodate them). The common ground of most of the approaches can be summarized as follows:

– create and manipulate data mining models through a SQL-based interface (thus implementing a *"command-driven"* data mining metaphor);
– abstract away the algorithmic particulars;

– allow for mining tasks to be performed on data in the database (thus avoiding the need to export to a special-purpose environment).

Approaches differ on what kinds of models should be created (which patterns are of interest), and what operations we should be able to perform (which constraints the patterns should satisfy). The query language proposed in [21,22] extends SQL with the new operator MINE RULE, which allows the computation and coding of associations in a relational format. Let us consider the relation transaction(Date, CustID, Item, Value) that contains the transactions of a sales representative. The following rule allows the extraction of the rules with support 20% and confidence 50%:

```
MINE RULE Associations AS
    SELECT DISTINCT 1..n Item AS BODY, 1..1 Item AS HEAD,
                        SUPPORT,CONFIDENCE
    WHERE BODY.Value > 100 AND HEAD.Value > 100
    FROM transaction
    GROUP BY CustID
        HAVING COUNT(Item) > 4
    CLUSTER BY Date
        HAVING BODY.Date < HEAD.Date
EXTRACTING RULES WITH SUPPORT: 0.2, CONFIDENCE: 0.5
```

The above expression specifies the mining of associations of purchased items such that the right part of the rule (consisting of only 1 item) has been purchased after the left part of the rule (that can consist of more than one item), and related to those customers who bought more than 4 items. Moreover, we consider only items with a value greater than 100.

The above approach reflects the following features:

– The source data is specified as a relational entity, and data preparation is accomplished by means of the usual relational operators. For example, the source table can be specified by means of usual join operations, selections and projections.
– The extended query language allows mining of unidimensional association rules. The GROUP BY keyword allows the specification of the transaction identifier, while the item description is specified in the SELECT part of the operator.
– Limited forms of background knowledge can be specified, by imposing some conditions over the admitted values of BODY and HEAD, and by using multiple source tables. Notice, however, that relying directly on SQL does not allow direct specification of more expressive constructs, such as, e.g., concept hierarchies. A limited form of data reorganization is specified by the CLUSTER keyword, that allows the specification of *topology* constraints (i.e. membership constraints of the components of rules to clusters).
– Concerning interestingness measures, the above operator allows the specification of the usual support and confidence constraints, and of further constraints over the contents of the rules (in particular, the SELECT keyword allows the specification of cardinality constraints).

– extracted knowledge is represented by means of relational tables, containing the specification of four attributes: Body, head, Support, Confidence.

Similarly to `MINE RULE`, the DMQL language [13,14], is designed as an extension of SQL that allows to select the primary source knowledge in SQL-like form. However, the emphasis here is on the kind of patterns to be extracted. Indeed, DMQL supports several mining tasks involving rules: characteristic, discriminant, classification and association rules. The following query:

```
use database university_database find characteristic rules
related to gpa, birth_place, address, count(*)%
from student where status = "graduate" and major = "cs"
and birth_place = "Canada"
with noise threshold = 0.05
```

specifies that the database used to extract the rules is the university database (`use database university_database`), that the kind of rules you are interested in are characteristic rules (`find characteristic rules`) w.r.t. the attributes gpa, birth place, and address (`related to ...`). The query specifies also that this rules are extracted on the students who are graduated in computer science, and that are born in Canada. As for MINE RULE, the specification of primary source knowledge is made explicit in the `from` and `where` clauses.

DMQL exploits follows a decoupled approach between specification and implementation, since the extraction is accomplished by external algorithms, and the specification of the query has the main objective of preparing the data and encoding them in a format suitable for the algorithms. Interestingly, DMQL allows the manipulation of a limited form of background knowledge, by allowing the direct specification of concept hierarchies.

Unfortunately, neither MINE RULE nor DMQL provide operators to further query the extracted patterns. The closure principle is marginally considered in MINE RULE (the mining result is stored into a relational table and can be further queried), but not considered at all within DMQL. By contrast, Imielinkski and others [17] propose a data mining query language (MSQL) which seeks to provide a language both to selectively generate patterns, and to separately query them. MSQL allows the extraction of association rules only, and can be seen as an extension of MINE RULE. The pattern language of MSQL is based on multidimensional propositional rules, which are specified by means of *descriptors*. A descriptor is an expression of the form: $(A_i = a_{ij})$ where A_i is an attribute, and a_{ij} is a either a value or a range of values in the domain of A_i. The rules extracted from MSQL have hence the form `Body` $\Rightarrow$ `Consequent` where `Body` is a conjunctset (i.e. the conjunction of an arbitrary number of descriptors such that each descriptor in the set refers to a different attribute) and `Consequent` is a single descriptor. Rules are generated by means of a `GetRules` statement which, apart from syntax issues, has similar features as MINE RULE and DMQL. In addition, MSQL allows for nested queries, that is, queries containing subqueries.

The extracted rules are stored in a *RuleBase* and then they can be further queried, by means of the `SelectRules` statement. It is possible to select a subset of the generated rules that verify a certain condition

```
SelectRules(R) where Body has { (Age=*), (Sex=*) } and Consequent
is { (Address=*) }
```

as well as to select the tuples of the input database that violate (satisfy) all (any of) the extracted rules:

```
Select * from Actor where VIOLATES ALL(
    GetRules(Actor)
    where Body is { (Age = *) }
    and Consequent is { (Sex = *) }
    and confidence > 0.3
        )
```

A novel and completely different perspective to inductive databases querying has been devised in [10]. The basic intuition is that, if the pattern language $\mathcal{L}$ were stored within relational tables, any constraint predicate Q could be specified by means of a relational algebra expression, and the DBMS could take care of implementing the best strategy for computing the solution space. Assume, for example, that sequences are stored within a relational engine by means of the following relations:

- Sequences(sid,item,pos), representing each sequence by means of a sequence identifier, an item and its relative position within the sequence;
- Supports(sid,supp) which specifies, for each sequence, its frequency.

Then, the following SQL query asks for the sequences holding with frequency greater than 60%, or such that item `a` occurs before item `b` within the transaction, can be expressed as follows:

```
SELECT Supports.sid
FROM Sequences S1, Sequences S2, Supports
WHERE S1.sid = Supports.sid AND S2.sid = S1.sid
  AND Supports.supp > 60
  OR (S1.item = a AND S2.item = b AND S1.pos < S2.pos)
```

Clearly, the pattern language can be extremely huge, and hence it is quite unpractical to effectively store it. Indeed, the pattern language is represented as a *virtual* table, i.e., an empty table which has to be populated. In the above example, although the `Sequences` and `Supports` tables are exploited within the query, they are assumed to be virtual tables, i.e., no materialization actually exists for them within the DBMS. The idea here is that, whenever the user queries such pattern tables, an efficient data mining algorithm is triggered by the DBMS, which materializes those tuples needed to answer the query. Afterwards, the query can be effectively executed. Thus, the core of the approach is a constraint extraction procedure, which analyzes a given SQL query and identifies the relevant constraints. The procedure builds, for each SQL query, the corresponding relational algebra tree. Since virtual tables appear in leaf nodes of the tree, a bottom-up traversal of the tree allows the detection of the necessary constraints. Finally, specific calls to a mining engine can be raised in order to populate those nodes representing virtual tables.

This approach has the merit of providing a real tight coupling between the mining and the DBMS, or in other terms, between the mining queries and the database queries. Indeed, this approach does not even require the definition of a data mining query language, since it is SQL itself to play such role. However, it is not clear how such approach could support a complex knowledge discovery process. For instance, the pre-processing step is completely overlooked by this approach: preparing the data for mining would require long and complex SQL queries. Moreover, since we got no reference to the source data, it is not clear how the mining view could be defined and/or changed within a mining session. Consider again the `Sequences` and `Supports` relations in the example above, and suppose that the support of sequences patterns are computed w.r.t. a database of sequences of events with a *weekly* timescale: what if the analyst decides to move to the *daily* timescale?

The problem of providing little support to the pre-processing and evaluation phase of the knowledge discovery process, is common to all the query languages discussed above. In ConQueSt, while we can take care of the pre-processing at the language level (e.g., easy mining view definition, attributes discretization), the evaluation phase is attacked merely at the system level by means of the pattern browser capabilities, such as the on-the-fly constraints tuning. More sophisticated post-processing of the extracted patterns, and reasoning techniques, should be studied and developed in our future work.

5 Conclusion

Many distinguishing features make ConQueSt a unique system:

Large variety of constraints handled - To the best of our knowledge, ConQueSt is the only system able to deal with so many different constraints all together, and provide the opportunity of easily defining new ones. While some prototypes for constraint-based pattern discovery exist, they are usually focused on few kinds of constraints, and their algorithmic techniques can not be easily extended to other constraints.

Usability - ConQueSt has been devised to fruitfully deal with real-world problems. The user friendly interface, the pre-processing capabilities and the simple connectivity to relational DBMS, make it easy for the user to immediately start to find nuggets of interesting knowledge in her/his data. Modularity and extensibility make the system able to adapt to changing application needs. Efficiency, robustness and scalability make possible to mine real-world huge datasets.

Robustness and resources awareness - One of the main drawbacks of the state-of-the art software for pattern discovery, is that it usually fails to mine large amounts of data due to memory lack. In this sense, ConQueStis robust, since huge datasets are mined out-of-core until the data-reduction framework reduces the dataset size enough to move it in-core.

Efficiency - ConQueSt is a high performance mining software. As an Example consider Figure 6 where we compare execution times of ConQueSt against

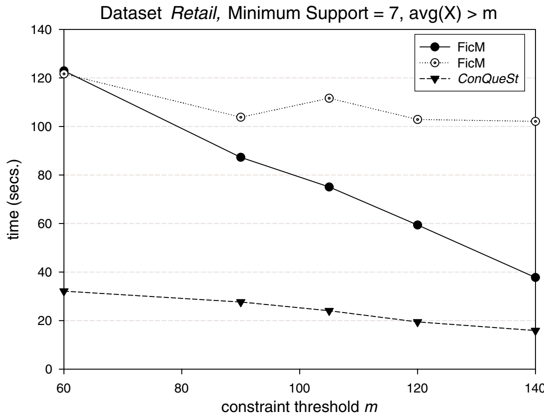


Fig. 6. Performance comparison

$\mathcal{FIC}^{\mathcal{A}}$ and $\mathcal{FIC}^{\mathcal{M}}$ [26], two depth first algorithms, ad-hoc devised to deal with the `avg` constraint (the one used in the comparison).

Even tough ConQueSt is already fruitfully usable on real-world problems, many direction must be explored in the next future: efficient incremental mining, advanced visualization techniques, more complex post-processing, building global models from the interesting patterns, mining patterns from complex data such as sequences and graphs. We are continuously developing new functionalities of ConQueSt.

References

1. http://java.sun.com/products/jdbc/
2. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of VLDB 1994 (1994)
3. Bistarelli, S., Bonchi, F.: Extending the soft constraint based mining paradigm. In: KDID 2006. LNCS, vol. 4747, pp. 24–41. Springer, Heidelberg (2007)
4. Bonchi, F.: Frequent Pattern Queries: Language and Optimizations. PhD thesis, Ph.D. thesis TD10- 03, Dipartimento di Informatica, Università di Pisa (2003)
5. Bonchi, F., Giannotti, F., Mazzanti, A., Pedreschi, D.: ExAMiner: Optimized level-wise frequent pattern mining with monotone constraints. In: Proceedings of ICDM 2003 (2003)
6. Bonchi, F., Giannotti, F., Mazzanti, A., Pedreschi, D.: ExAnte: Anticipated data reduction in constrained pattern mining. In: Lavrač, N., Gamberger, D., Todorovski, L., Blockeel, H. (eds.) PKDD 2003. LNCS (LNAI), vol. 2838, Springer, Heidelberg (2003)
7. Bonchi, F., Lucchese, C.: On closed constrained frequent pattern mining. In: Perner, P. (ed.) ICDM 2004. LNCS (LNAI), vol. 3275, Springer, Heidelberg (2004)

8. Bonchi, F., Lucchese, C.: Pushing tougher constraints in frequent pattern mining. In: Ho, T.-B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, Springer, Heidelberg (2005)
9. Bucila, C., Gehrke, J., Kifer, D., White, W.: DualMiner: A dual-pruning algorithm for itemsets with constraints. In: Proceedings of ACM SIGKDD 2002, ACM Press, New York (2002)
10. Calders, T., Goetals, B., Prado, A.: Integrating pattern mining in relational databases. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 454–461. Springer, Heidelberg (2006)
11. Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P.: The kdd process for extracting useful knowledge from volumes of data. Commun. ACM 39(11), 27–34 (1996)
12. Grahne, G., Lakshmanan, L.V.S., Wang, X., Xie, M.H.: On dual mining: From patterns to circumstances, and back. In: Proceedings of the 17th International Conference on Data Engineering (ICDE 2001), April 2-6, 2001, Heidelberg, Germany (2001)
13. Han, J., Fu, Y., Koperski, K., Wang, W., Zaiane, O.: DMQL: A Data Mining Query Language for Relational Databases. In: SIGMOD 1996 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD 1996) (1996)
14. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufman, San Francisco (2000)
15. Han, J., Lakshmanan, L.V.S., Ng, R.T.: Constraint-based, multidimensional data mining. Computer 32(8), 46–50 (1999)
16. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Comm. Of The Acm 39, 58–64 (1996)
17. Imielinski, T., Virmani, A.: MSQL: A Query Language for Database Mining. Data Mining and Knowledge Discovery 3(4), 373–408 (1999)
18. Kramer, S., Raedt, L.D., Helma, C.: Molecular feature mining in hiv data. In: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, August 26-29, 2001, pp. 136–143. ACM Press, New York (2001)
19. Lakshmanan, L.V.S., Ng, R.T., Han, J., Pang, A.: Optimization of constrained frequent set queries with 2-variable constraints. SIGMOD Record 28(2) (1999)
20. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery 1(3), 241–258 (1997)
21. Meo, R., Psaila, G., Ceri, S.: A new SQL-like operator for mining association rules. In: Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L. (eds.) VLDB 1996, Proceedings of 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India, 3–6 september 1996, pp. 122–133. Morgan Kaufmann, San Francisco (1996)
22. Meo, R., Psaila, G., Ceri, S.: A Tightly-Coupled Architecture for Data Mining. In: International Conference on Data Engineering (ICDE 1998), pp. 316–323 (1998)
23. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained associations rules. In: Proceedings of the ACM SIGMOD 1998, ACM Press, New York (1998)
24. Orlando, S., Palmerini, P., Perego, R., Silvestri, F.: Adaptive and Resource-Aware Mining of Frequent Sets. In: Proc. of the 2002 IEEE Int. Conference on Data Mining (ICDM 2002), Maebashi City, Japan, December 2002, pp. 338–345. IEEE Computer Society Press, Los Alamitos (2002)

25. Pei, J., Han, J.: Can we push more constraints into frequent pattern mining? In: Proceedings of ACM SIGKDD 2000, ACM Press, New York (2000)
26. Pei, J., Han, J., Lakshmanan, L.V.S.: Mining frequent item sets with convertible constraints. In: Proceedings of ICDE 2001 (2001)
27. Esposito, R., Meo, R., Botta, M.: Answering constraint-based mining queries on itemsets using previous materialized results. Journal of Intelligent Information Systems (2005)
28. Srikant, R., Vu, Q., Agrawal, R.: Mining association rules with item constraints. In: Proceedings of ACM SIGKDD 1997, ACM Press, New York (1997)