RQL: A Query Language for Rule Discovery in Databases
Brice Chardin, Emmanuel Coquery, Marie Pailloux, Jean-Marc Petit

To cite this version:

HAL Id: hal-01395083
https://hal.archives-ouvertes.fr/hal-01395083
Submitted on 10 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
**RQL: A Query Language for Rule Discovery in Databases**

Brice Chardin¹, Emmanuel Coquery², Marie Pailloux³, Jean-Marc Petit⁴

---

**Abstract**

Promoting declarative approaches in data mining is a long standing theme, the main idea being to simplify as much as possible the way data analysts interact with their data. This paper goes into this direction by proposing a well-founded logical query language, \( \text{SafeRL} \), allowing the expression of a wide variety of rules to be discovered against a database. By rules, we mean statements of the form "if ... then ...", as defined in logics for "implications" between boolean variables. As a consequence, \( \text{SafeRL} \) extends and generalizes functional dependencies to new and unexpected rules. We provide a query rewriting technique and a constructive proof of the main query equivalence theorem, leading to an efficient query processing technique. From \( \text{SafeRL} \), we have devised RQL, a user-friendly SQL-like query language. We have shown how a tight integration can be performed on top of any relational database management system. Every RQL query turns out to be seen as a query processing problem, instead of a particular rule mining problem. This approach has been implemented and experimented on sensor network data. A web prototype has been released and is freely available (\( \text{http://rql.insa-lyon.fr} \)). Data analysts can upload a sample of their data, write their own RQL queries and get answers to know whether or not a rule holds (if not, a counter example from the database is displayed) and much more.

**keywords** Query Languages, Formal Concept Analysis, Implications, Functional dependencies, Query Optimization, Relational Calculus

1. Introduction

The relational database management systems (DBMS) market is already huge and continues to grow since it is expected to nearly double by 2016 [41]. As a trivial consequence for the data mining community, it makes sense – more
than ever – to query the data in-place when using state of the art database technologies.

While a lot of techniques have been proposed over the last 20 years for pattern mining, only a few of them are tightly coupled with a DBMS. Most of the time, some pre-processing has to be performed before the use of pattern mining techniques and the data have to be formatted and exchanged between different systems, turning round-trip engineering into a nightmare.

In this paper, we provide a logical view for a certain class of pattern mining problems. More precisely, we propose a well-founded logical query language, SafeRL, based on tuple relational calculus (TRC), allowing the expression of a wide variety of rules to be discovered against the data. By rules, we mean statements of the form "if ... then ...", as defined in logics for "implications" between boolean variables.

From a database perspective, SafeRL extends and generalizes functional dependencies (FDs) to new and unexpected rules easily expressed with a practical SQL-like language, called RQL, derived from SafeRL. To start with, let us consider the running example given in Figure 1 with a relation Emp to depict employees. Educlevel represents the number of years of formal education, Sal the yearly salary, Bonus the yearly bonus and Comm the yearly commission. This example will be used throughout the paper.

Intuitively, a RQL query is defined by the FINDRULES clause and generates rules of the form $X \rightarrow Y$ with $X$ and $Y$ disjoint attribute sets taken from the OVER clause. The SCOPE clause defines tuple-variables over some relations obtained by classical SQL queries and the CONDITION clause defines the predicate to be evaluated on each attribute, denoted by the named variable $\$A$, occurring in $X \cup Y$.

Example 1. To make things concrete, we give some examples of RQL queries.

Q1: FINDRULES

OVER Empno, Lastname, Workdept, Job, Sex, Bonus
SCOPE t1, t2 Emp
CONDITION ON $\$A IS t1.$A = t2.$A

Figure 1: Running example
$Q'_1$: \texttt{FINDRULES}
\begin{align*}
&\text{OVER Empno, Lastname, Workdept, Job, Sex, Bonus} \\
&\text{SCOPE t1, t2} ( \\
&\text{SELECT * FROM Emp WHERE Educlevel > 16} ) \\
&\text{CONDITION ON $A$ IS t1.$A = t2.$A}
\end{align*}

$Q''_1$: \texttt{FINDRULES}
\begin{align*}
&\text{OVER Educlevel, Sal, Bonus, Comm} \\
&\text{SCOPE t1, t2 Emp} \\
&\text{CONDITION ON $A$ IS} \\
&2\cdot\text{ABS}(t1.$A-t2.$A)/(t1.$A+t2.$A) < 0.1
\end{align*}

$Q_1$ discovers FDs from $Emp$ over the subset of attributes specified in the \texttt{OVER} clause. Recall that a FD $X \rightarrow Y$ holds in a relation $r$ if for all tuples $t1, t2 \in r$, and for all $A \in X$ such that $t1[A] = t2[A]$ then for all $A \in Y$, $t1[A] = t2[A]$. As shown in $Q_1$, RQL allows to express the discovery of FDs with a natural syntax. For example, $\text{Empno} \rightarrow \text{Lastname}$ and $\text{Workdept} \rightarrow \text{Job}$ hold in $Emp$.

We can easily restrict FDs to some subset of tuples as shown with $Q'_1$ which discovers rules comparable to conditional functional dependencies [14] by considering only employees with a level of qualification above 16. For instance, $\text{Sex} \rightarrow \text{Bonus}$ holds, meaning that above a certain level of qualification (16), the gender determines the bonus. This rule was not elicited by $Q_1$ because there exist several counter-examples, such as employees 30 and 70.

$Q''_1$ is an approximation of FDs for numeric values, similar to Metric Functional Dependencies [35], where strict equality is discarded to take into account variations under 10%. For instance, salaries 41250 and 38250 are considered close (7.5% difference), but not salaries 41250 and 36170 (13.1% difference). $\text{Sal} \rightarrow \text{Comm}$ then holds, meaning that employees earning similar salaries receive similar commissions.

With respect to the expressiveness of the language, RQL allows to ”catch” implications in their purest logic form. On the one hand, it turns out that two famous examples, namely association rules with 100% confidence and functional dependencies, can be expressed with RQL, since both are logical implications. Nevertheless, RQL can do much more, as shown in previous examples and throughout the paper (see section 1.1.). On the other hand, RQL cannot be used to express every possible rule mining problem. For instance, it has not been conceived to be equivalent to association rules, which are not strictly logical implications. Frequency requirements (e.g. support and confidence) immediately remove key properties such as reflexivity ($Y \subset X, X \rightarrow Y$ does not hold if $X$ is not frequent).

RQL has been devised as a user-friendly SQL-like query language, which can be integrated on top of any DBMS supporting SQL. RQL query processing is seen as a classical query processing problem in databases. We provide a query rewriting technique and a constructive proof of the main query equivalence theorem, leading to an efficient query processing technique.
This approach has been implemented and experimentally evaluated on sensor network data. A web prototype has been released and is freely available (http://rql.insa-lyon.fr). Data analysts can upload a sample of their data, write their own RQL queries and get answers to know whether or not a rule holds (if not, a counter example from the database is displayed) and much more.

This contribution is an attempt to bridge the gap between pattern mining and databases to facilitate the use of data mining techniques by SQL-aware analysts. The ultimate goal of this work is to integrate pattern mining techniques into core DBMS technologies.

1.1. More RQL examples

Conveniently, we have reused so far RQL examples related to FDs. Nevertheless, RQL does much more and is not restricted to FDs at all.

Example 2. null values in Dept.

Q2: FINDRULES
OVER Empno, Lastname, Workdept, Job, Sex, Bonus,
Mgrno
SCOPE t1 Emp
CONDITION ON $A IS t1.$A IS NULL

Q2 discovers rules between null values of the relation Emp. In most databases, null values are common and knowing relationships between attributes with respect to null values could be useful. For instance, Mgrno \(\rightarrow\) Workdept holds in Emp, meaning that when Mgrno is null, then Workdept is also null for the same tuple (only employee No. 20 in example 1).

Example 3. Assume we are interested in a kind of sequential dependencies [27], i.e. dependencies showing similar behavior of attribute values. Q3 discovers numerical attributes that vary together (i.e., \(X \rightarrow Y\) means that if \(X\) increases then \(Y\) also increases).

Q3: FINDRULES
OVER Educlevel, Sal, Bonus, Comm
SCOPE t1, t2 Emp
CONDITION ON $A IS t1.$A >= t2.$A

Using Q3, Sal \(\rightarrow\) Comm and Comm \(\rightarrow\) Sal hold in Emp, which means that a higher salary is equivalent to a higher commission.

Example 4. Tuple-variables can also be defined on different relations. For instance, the following query searches for inequalities between tuple-variables from two views, referred to as managers and managees in the query.

FINDRULES OVER Educlevel, Sal, Bonus, Comm
SCOPE
managers (SELECT * FROM Emp WHERE Empno IN (}
SELECT Mgrno FROM Emp

managees (SELECT * FROM Emp WHERE Empno NOT IN (SELECT Mgrno FROM Emp WHERE Mgrno IS NOT NULL))

CONDITION ON $A IS managers.$A > managees.$A

The rule $0 \rightarrow \text{Educlevel}$ then holds, meaning that managers always have a higher education levels than non-managers.

The interested reader may find more intricate examples in [18].

1.2. RQL query processing in a nutshell

This section unfolds one of the provided examples to convey the core ideas underlying RQL query processing, for which a general architecture is given in figure 3, section 5. To do so, we consider the following query $Q$ (simplified from query $Q_1$ given in Example 1):

$Q$: FINDRULES
OVER Empno, Job, Sex, Bonus
SCOPE t1, t2 (SELECT * FROM Emp WHERE Educlevel > 16)
CONDITION ON $A IS t1.$A = t2.$A

After the usual lexical and syntactic analysis, RQL query processing is composed of two main steps:

1. Computing, against the database, a representation of the set of rules represented by the query. Such a representation is a set of attribute sets called a base of the associated closure system.

2. Computing a cover of the set of rules from this base.

The first step builds a base with respect to the condition specified in the CONDITION clause of the query. The basic idea is to perform the necessary joins and cross products between the relations of the SCOPE clause, and to apply a filter to elicit attribute sets satisfying the criteria.

Interestingly, a single SQL query is enough to compute such a base. With Oracle DBMS, conditions are specified using conditional statements (CASE WHEN) and then concatenated (|| operator) to provide a string representation of attribute sets. Note that such operations are simple enough to be present in every DBMS.

For the previous query $Q$, the RQL query processor generates the following SQL query, whose output is given in Table 1. This result is no more than a binary relation (called a context), well known in the Formal Concept Analysis (FCA) community.
Table 1: Base of $Q$ in $Emp$

SELECT DISTINCT (
    CASE WHEN t1.Empno = t2.Empno THEN 'Empno ' END |
    CASE WHEN t1.Job = t2.Job THEN 'Job ' END ||
    CASE WHEN t1.Sex = t2.Sex THEN 'Sex ' END ||
    CASE WHEN t1.Bonus = t2.Bonus THEN 'Bonus ' END
) as base
FROM (SELECT * FROM Emp WHERE Educlevel > 16) t1,
     (SELECT * FROM Emp WHERE Educlevel > 16) t2

From such a base, the RQL query processor may achieve the second step – computing a cover of the set of rules satisfied in the database with respect to $Q$ – for which we mainly reused well-known techniques from database and FCA (see sections 1.3 and 2.4). This cover consists of three functional dependencies:

- $Empno \rightarrow Job\ Sex\ Bonus$
- $Sex \rightarrow Bonus$
- $Bonus \rightarrow Sex$

As a matter of fact, the base also serves as a starting point to compute the closure of an attribute set and to decide whether or not a given rule is satisfied. These computations are outlined in Section 2.4.

1.3. Related work

Declarative and generic approaches for pattern mining have been studied by many researchers under different angles (databases, constraint programming or theoretical foundations), see for example [33, 40, 7, 24, 15, 31, 47, 46, 48]. A crucial question underlying all these approaches is about the languages to first specify the data of interest and second, to express patterns of interest. A survey of data mining query languages has been done in [12] and the cross-fertilization between data mining and constraint programming has been investigated recently in [31, 44].

Nevertheless, we argue that pattern mining languages should benefit from direct extensions of SQL languages, since data are most of the time stored in DBMSs. Practical approaches, as close as possible to DBMSs, have been proposed for example in [43, 22, 49, 13] to interact more directly with DBMSs query engines. Nevertheless, as far as we know, we are not aware of systems similar to RQL since it targets a very specific class of rules, i.e. those rules verifying Armstrong axioms and thus classical implication in logics. In other words, association rules systems such as MINE RULE [43] or data mining extensions of SQL systems (e.g. Oracle, IBM) cannot be fairly compared to RQL. Moreover,
RQL cannot be compared to association rules systems due to the frequency requirement.

The SafeRL language, inspired from the logical language proposed in [15], goes into this direction by providing a formal semantic based on the tuple relational calculus (TRC), underpinning SQL. FDs, association rules with 100% confidence and the ad-hoc language proposed in [6] are special cases of our SafeRL language but none of them has a logical query language foundation. In this paper, we provide a logical view on the ad-hoc language proposed in [6] for gene expression data. The language of [6] was defined using a BNF grammar with no clear logical foundation and hence no practical SQL-like query language. From a technical point of view, the paper can be seen as a generalization of the approach proposed in [39, 20, 38] to discover functional dependencies (FD). Indeed, the machinery introduced in this paper generalizes the computation of agree sets (a base of the closure system associated with FD) to every RQL query. Moreover, the paper is a major extension of [5, 17]: [5] provides a logical view on the ad-hoc language proposed in [6], with some restrictions (e.g. only one relation schema). Theoretical results on decidability problems are also given. We do not reproduce all the results in this paper. [17] introduces the practical query language RQL for the first time. Moreover, RQL has been used as a core building block to build a complete suite for genomics-data analysis in the RulNet prototype [52]. In RulNet, RQL is used with a slightly different syntax but with the same query engine to process the queries. Many options are provided to conveniently display a set of rules as gene regulatory network and to assess the quality of rules with different interestingness measures. Interestingly, other applications such as [3] exist for data analysis in biomedical studies with implications, for which RQL could also be useful.

Many dependencies different from functional dependencies have been studied such as implications in formal concept analysis (FCA) [38, 42, 9], conditional FDs [14], sequential dependencies [27], metric FDs [35], denial constraints [19] or constraint-generating dependencies [10]. They can partially be represented with RQL (cf. examples in Section 1) with some restrictions though. Nevertheless, RQL widens the scope of dependencies and lets the data analysts decide their patterns of interest with respect to their background knowledge, without any presupposition on the dependencies to be discovered.

1.4. Paper organization

Section 2 introduces some notations and recalls important notions on relational calculus and closure systems. Section 3 presents the syntax and semantics of the SafeRL language, while section 4 presents some results used for computing the answer to SafeRL queries. Section 5 presents experimental results, section 6 presents the web prototype for RQL and section 7 concludes.

2. Preliminaries

This section introduces main definitions and notations used throughout the paper for the relational model, safe TRC, rules and closure systems.
2.1. Relational model

We use the named perspective of the relational model in which tuples are functions \([1]\).

Fix a finite universe \(U\) of attributes (denoted by \(A, B, \ldots\)), a countably infinite domain \(D\) of constants (denoted by \(c, c', \ldots\)) and a finite set \(R\) of relation symbols (denoted by \(R, S, \ldots\)). \(U, D, R\) are pairwise disjoint. Each relation symbol \(R\) has a schema, a subset of \(U\), denoted by the symbol itself, i.e. \(R \subseteq U\). Conveniently, we will sometimes omit to refer to the relation symbol when dealing with a subset of attributes, i.e. a schema. A tuple \(t\) over \(R\) is a total function \(t: R \rightarrow D\). A relation \(r\) over \(R\) is a finite set of tuples over \(R\). A database schema \(R\) is a set of relation symbols, e.g. \(R = \{R_1, \ldots, R_n\}\). A database instance (or simply a database) is a function \(d\) from \(R\) to the set of possible relations such that \(d(R_i) = r_i\), with \(r_i\) a relation over \(R_i\) for \(i = 1..n\).

2.2. Variables and assignments

Safe \(RL\) has different formal variables for attributes, tuples and schemata: a set \(A\) of attribute-variables \((A, B, \ldots)\), a set \(T\) of tuple-variables \((s, t, \ldots)\) and a set \(S\) of schema-variables \((X, Y, \ldots)\). \(A, T, S, U, D, R\) are pairwise disjoint.

An attribute-assignment \(\rho\) (resp. a schema-assignment \(\Sigma\)) is a function that maps an attribute-variable \(A\) (resp. a schema-variable \(X\)) to an attribute \(\rho(A) \in U\) (resp. a subset of attributes \(\Sigma(X) \subseteq U\)). A tuple-assignment \(\sigma\) is also a function from a tuple-variable \(t\) to a tuple \(t\) defined over some schema. Conveniently, a tuple-variable \(t\) can be explicitly defined over some schema \(X\), noted by \(t: X\) and we will use the notation \(sch(t) = X\).

For an attribute-assignment \(\rho\) (as well as for tuple-assignments and schema-assignments) we denote by \(\rho_{A \mapsto A}\) the assignment defined by:

\[
\rho_{A \mapsto A}(B) = \begin{cases} 
A & \text{if } B = A \\
\rho(B) & \text{if } B \neq A
\end{cases}
\]

2.3. Safe TRC

Since TRC is a core component of Safe \(RL\), we recall here the syntax and semantics of the TRC in its simplest form (see [1] for more details). TRC formulas noted \(\psi, \psi_1, \psi_2, \ldots\) are defined inductively as usual, where \(A, B \in U, X \subseteq U, c \in D, R \in R, t, t_1, t_2 \in T:\)

\[
R(t) \mid t_1.A = t_2.B \mid t.A = c \mid \neg \psi \mid \psi_1 \land \psi_2 \mid \exists t : X (\psi)
\]

Given a database \(d\) over \(R\) and a tuple assignment \(\sigma\), the satisfaction of a TRC formula \(\psi\) is inductively defined as follows:

- \(\langle d, \sigma \rangle \models R(t)\) if \(\sigma(t) \in d(R), R \in R\)
- \(\langle d, \sigma \rangle \models t_1.A = t_2.B\) if \(\sigma(t_1)(A) = \sigma(t_2)(B)\)
- \(\langle d, \sigma \rangle \models t.A = c\) if \(\sigma(t)(A) = c\)
• \( \langle d, \sigma \rangle \models \neg \psi \) if \( \langle d, \sigma \rangle \not\models \psi \)

• \( \langle d, \sigma \rangle \models \psi_1 \land \psi_2 \) if \( \langle d, \sigma \rangle \models \psi_1 \) and \( \langle d, \sigma \rangle \models \psi_2 \)

• \( \langle d, \sigma \rangle \models \exists t : X (\psi) \) if there exists a tuple \( t \) over \( X \) such that \( \langle d, \sigma_{t \rightarrow t} \rangle \models \psi \)

A TRC query is an expression of the form

\[ q = \{ t \mid \psi \} \]

where \( \psi \) is a TRC formula with exactly one free variable \( t \). The set of answers \( \text{ans}(q, d) \) of \( q \) w.r.t. a database \( d \) is

\[ \text{ans}(q, d) = \{ \sigma(t) \mid \langle d, \sigma \rangle \models \psi \} \]

In the sequel, we consider safe TRC, the fragment of TRC known to always provide finite answers [1].

Moreover, we shall admit several tuple variables in TRC formulas: Let \( \psi(t_1, \ldots, t_n) \) be a safe TRC formulas with \( t_1, \ldots, t_n \) free variables, i.e. \( q = \{ \langle t_1, \ldots, t_n \rangle \mid \psi(t_1, \ldots, t_n) \} \). Then the answer is just a set of tuple assignment \( \sigma \) defined for each \( t_i, i = 1..n \), i.e. \( \text{ans}(q, d) = \{ \sigma_{t_1,\ldots,t_n} \mid \langle d, \sigma \rangle \models \psi(t_1, \ldots, t_n) \} \).

2.4. Rules and closure systems

Rules or implications, closure systems and closure operators have been widely studied in many branches of applied mathematics and computer sciences, with applications in databases for functional dependencies [8] and in formal concept analysis for implications [26]. The interested reader should refer to [16, 11] for a comprehensive survey. We summarize the main results that are useful for the rest of the paper.

Let \( U \subseteq \mathbb{U} \) and \( X, Y \subseteq U \). A rule is a syntactic expression of the form \( X \rightarrow Y \). \( C \subseteq 2^U \) is a closure system if \( U \in C \) and \( X, Y \in C \Rightarrow X \cap Y \in C \) [26].

Let \( F \) be a set of rules on \( U \). A rule \( X \rightarrow A \) is implied by \( F \), denoted by \( F \vdash X \rightarrow A \), if there is a derivation (or a proof) from \( F \) using Armstrong’s axiom system (reflexivity, augmentation, transitivity) ending to \( X \rightarrow A \). The closure of a set \( X \subseteq U \) is defined by \( X^+ \) \( F = \{ A \mid F \vdash X \rightarrow A \} \). A closure system can be defined for \( F \), noted \( CL(F) = \{ X \subseteq U \mid X = X^+ \} \).

Let \( IRR(F) \) be the set of meet-irreducible elements of \( CL(F) \), i.e. \( X \in IRR(F) \) iff all \( Y, Z \in CL(F), (X = Y \cap Z) \Rightarrow (X = Y \text{ or } X = Z) \).

The notion of base of a closure system is now defined as follows:

**Definition 1.** Let \( CL(F) \) be a closure system. A base \( B \) of \( CL(F) \) is such that \( IRR(F) \subseteq B \subseteq CL(F) \)

A base is called a context in FCA terminology [26].

It is worth noting that whenever a base has been computed from a given relation \( r \), we can address the following problems:

1. Given a set of attributes, compute its closure with respect to the rules satisfied in \( r \).
2. Given a rule, say whether or not this rule is satisfied. If not, give a counter-example from \( r \).

3. Compute a cover of non satisfied rules in \( r \).

4. Compute a cover (or basis) of satisfied rules in \( r \).

Let us consider the well-known functional dependencies. Given a relation \( r \) over \( R \), a base of the closure system associated to satisfied FDs in \( r \) is known as the set of agree sets. Given the agree sets \( ag(r) \), each problem listed above has been studied extensively in database and formal concept analysis communities [38]:

1. Compute the closure of a set of attributes: let \( X \subseteq R \). \( X^+ = \bigcap \{ Y \in ag(r) | X \subseteq Y \} \).

2. Verify a rule: let \( X \rightarrow Y \) be a rule, \( r \models X \rightarrow Y \) iff \( Y \subseteq X^+ \).

3. Compute a cover of approximate FDs, known as the Gottlob and Libkin cover [29]: \( X \rightarrow A \) is given whenever \( X \in \max(A, r) = \max \subseteq \{ X \in ag(r) | A \not\in X \} \).

4. Compute the canonical cover\(^5\) for satisfied FDs [39, 20, 38]: \( X \rightarrow A \) is given whenever \( X \) is a minimal transversal of the hypergraph \( \{ R \setminus X | X \in \max(A, r) \} \).

From a complexity point of view, steps 1 to 3 described above are polynomial while the fourth one is incremental quasi-polynomial in the size of the input and the output due to the enumeration of minimal transversal [25].

The rest of this paper proposes a generalization of this approach. Indeed, each SafeRL query defines a closure system and therefore, in order to reuse previous results, the problem turns out to be on the computation against the database of a base with respect to a given SafeRL query. In this paper, we will focus on this problem only. Other problems listed above such as the generation of a canonical cover will not be detailed.

3. A Query Language for Rule Mining

In the introduction, we have illustrated RQL – an SQL-like friendly language – through examples. This section formally defines the syntax and semantics of SafeRL from which RQL is derived. We have introduced safe TRC for expressing SQL-like queries. Before defining SafeRL, it remains to precisely define the notion of mining formulas, denoted in RQL (cf. previous examples) as:

\[
\text{CONDITION ON } \$A \text{ IS } \delta(\$A, t_1, \ldots, t_n)
\]

\(^5\)Also called canonical direct basis [11]
3.1. Mining Formulas

Mining formulas, denoted by $\delta, \delta_1, \delta_2, \ldots$, are defined over tuple-variables $T$, attribute-variables $A$ and constants $D$ only. Their syntax and their semantics are defined as follows.

**Definition 2.** Let $t, t_1, t_2 \in T$, $A, B \in A$ and $c \in D$. A mining formula is of the form:

$t_1.A = t_2.B$ | $t.A = c$ | $\neg \delta$ | $\delta_1 \land \delta_2$ where $\delta, \delta_1, \delta_2$ are mining formulas.

The satisfaction of a mining formula $\delta$ w.r.t. a tuple-assignment $\sigma$ and an attribute-assignment $\rho$, denoted by $\langle \sigma, \rho \rangle |\delta|$, is inductively defined as follows:

- $\langle \sigma, \rho \rangle | t_1.A = t_2.B | \sigma(t_1)(\rho(A)) = \sigma(t_2)(\rho(B))$
- $\langle \sigma, \rho \rangle | t.A = c | \sigma(t)(\rho(A)) = c$
- $\langle \sigma, \rho \rangle | \neg \delta | \neg \sigma(\rho)$
- $\langle \sigma, \rho \rangle | \delta_1 \land \delta_2 | \langle \sigma, \rho \rangle |\delta_1 | \land \langle \sigma, \rho \rangle |\delta_2$

Such formulas are very simple and relatively restrictive to keep the presentation simple. In practice, we shall use other binary operators such as $\leq, <, \geq, >, \ldots$ Details are omitted.

3.2. SafeRL queries

The SafeRL query language can now be defined.

**Definition 3.** A SafeRL query over a database schema $R$ is an expression of the form:

$$Q = \{ X \rightarrow Y \mid \forall t_1 \ldots \forall t_n [\psi(t_1, \ldots, t_n) \rightarrow \forall A \in X(\delta(A, t_1, \ldots, t_n)) \rightarrow \forall A \in Y(\delta(A, t_1, \ldots, t_n))] \}$$

where:

- $X$ and $Y$ are schema-variables,
- $\psi$ is a TRC-formula over $R$ with $n$ free tuple-variables $t_1, \ldots, t_n$,
- $\delta$ is a mining formula with $t_1, \ldots, t_n$ free tuple-variables and $A$ a single free attribute-variable.

When clear from context, a SafeRL query $Q$ may also be simply denoted by $Q = \langle \psi(t_1, \ldots, t_n), \delta(A, t_1, \ldots, t_n) \rangle$, or even $Q = \langle \psi, \delta \rangle$.

**Example 5.** Continuing example 1, query $Q_1$ is formalized in SafeRL as follows:

$$Q_1 = \{ X \rightarrow Y \mid \forall t_1, t_2 [(\text{EMP}(t_1) \land \text{EMP}(t_2)) \rightarrow \forall A \in X(t_1.A = t_2.A) \rightarrow \forall A \in Y(t_1.A = t_2.A)] \}$$

More succinctly, $Q_1$ is also noted $\langle \text{EMP}(t_1) \land \text{EMP}(t_2), (t_1.A = t_2.A) \rangle$. 

11
The attributes appearing in the result of $\psi$ are equal to $\bigcup_{i=1}^{n} sch(t_i)$ whereas the schema of $Q$, denoted by $sch(Q)$, is defined by: $sch(Q) = \bigcap_{i=1}^{n} sch(t_i)$. Indeed, only common attributes of tuple-variables are meaningful to discover rules.

To specify the result of the evaluation of a $SafeRL$ query against a database, we define the notion of satisfaction.

**Definition 4.** A $SafeRL$ query $\langle \psi, \delta \rangle$ is satisfied in a database $d$ and w.r.t. a schema-assignment $\Sigma$, denoted by $\langle d, \Sigma \rangle |\!|=\!\langle \psi, \delta \rangle$, if the following holds:

\begin{itemize}
  \item[(1)] For all tuple-assignment $\sigma$ such that $\langle d, \sigma \rangle |\!|=\!\psi$:
  \begin{itemize}
    \item if for all $A \in \Sigma(X)$, $\langle \sigma, \rho_{A \rightarrow A} \rangle |\!|=\!\delta$
    \item then for all $A \in \Sigma(Y)$, $\langle \sigma, \rho_{A \rightarrow A} \rangle |\!|=\!\delta$
  \end{itemize}
\end{itemize}

Intuitively, this definition generalizes the definition of FD satisfaction in a relation: instead of only 2 tuples in a relation, we may have $n$ tuples from the database $d$ satisfying $\psi$ (cf (1)); and instead of the condition "for all $A \in X, t_1[A] = t_2[A]\"", we have "for all $A \in \Sigma(X), \delta(A, t_1, \ldots, t_n)\"" (cf. (2) and (3)).

**Definition 5.** The answer of a $SafeRL$ query $Q = \langle \psi, \delta \rangle$ in a database $d$ over $R$, denoted by $\text{ans}(Q, d)$, is defined as:

\[
\text{ans}(Q, d) = \{ \Sigma(X) \rightarrow \Sigma(Y) \mid \langle d, \Sigma \rangle |\!|=\!\langle \psi, \delta \rangle, \Sigma(X) \cup \Sigma(Y) \subseteq sch(Q) \}
\]

### 3.3. RQL: A practical language for $SafeRL$

RQL is a practical SQL-like declarative language to express $SafeRL$ queries. Let us consider a $SafeRL$ query $Q = \langle \psi(t_1, \ldots, t_n), \delta(A, t_1, \ldots, t_n) \rangle$ and its associated RQL query:

```sql
FINDRULES
OVER $A_1, \ldots, A_n$
SCOPE $t_1$ (SQL1), $t_n$ (SQLn)
WHERE condition($t_1, \ldots, t_n$)
CONDITION ON $A$ IS $\delta(A, t_1, \ldots, t_n)$
```

The `FINDRULES` clause identifies RQL queries. The `OVER` clause specifies the set of attributes to be used for rule discovery. Those attributes have to be included in $sch(Q)$. The `SCOPE` clause specifies every tuple to be used to discover the rules and corresponds to the tuple-variables of $\psi$. Each tuple-variable is associated with an SQL query (which can be factorized if they refer to the same SQL query). The `WHERE` clause is borrowed from the SQL `WHERE` clause to specify relationships between tuple variables. The `CONDITION ON $A$ IS $\delta$` specifies the mining formula $\delta$.

Note that RQL allows much more flexibility than $SafeRL$ since more advanced conditions available in SQL – such as regular expressions, user-defined functions or built-in functions of the underlying DBMS – can be used for free, as in query $Q'_1$ of Example 1.
4. Theoretical results

In [5], a slightly different language for rule mining has been proposed. One of the main results was to point out that every query was "Armstrong-compliant", meaning basically that Armstrong axioms are sound and that each query defines a closure system. The same result holds for SafeRL queries as we shall see. This result means that for query processing, we can adopt a strategy that generalizes the process of FD inference through agree sets [39, 37].

Given a database \(d\) and a SafeRL query \(Q\), the basic idea is to compute a base of the closure system associated with \(Q\) from \(d\). Let us start by introducing the closure system associated with \(Q\), then the notion of base.

4.1. Closure system and bases for SafeRL queries

Given a query \(Q\) against a database \(d\), the definitions of a base and a closure system given in Section 2.4 are extended to \(\text{ans}(Q, d)\).

**Definition 6.** We say that \(Z \subseteq U\) satisfies \(\text{ans}(Q, d)\) if for all \(X \rightarrow Y \in \text{ans}(Q, d)\), \(X \not\subseteq Z\) or \(Y \subseteq Z\). The closure system of \(Q\) in \(d\), denoted by \(\text{CL}_Q(d)\), is defined by: \(\text{CL}_Q(d) = \{Z \subseteq \text{sch}(Q) \mid Z\ \text{satisfies}\ \text{ans}(Q, d)\}\).

**Lemma 1.** Let \(Q\) be a SafeRL query and \(d\) a database. Then, \(\text{CL}_Q(d)\) is a closure system on \(\text{sch}(Q)\).

**Proof 1.** We have to show that \(\text{sch}(Q) \in \text{CL}_Q(d)\) and for all \(X, Y \in \text{CL}_Q(d)\), \(X \cap Y \in \text{CL}_Q(d)\).

- \(\text{sch}(Q)\) satisfies \(\text{ans}(Q, d)\) since for every \(X \rightarrow Y \in \text{ans}(Q, d)\), we have \(Y \subseteq \text{sch}(Q)\).

- Let \(X, Y \in \text{CL}_Q(d)\). We have to show that \(X \cap Y \in \text{CL}_Q(d)\), i.e. \(X \cap Y\) satisfies \(\text{ans}(Q, d)\). By hypothesis, we have \(X \in \text{CL}_Q(d)\) and \(Y \in \text{CL}_Q(d)\), implying for every \(V \rightarrow W \in \text{ans}(Q, d)\), \((V \not\subseteq X\ or\ W \subseteq X)\) and \((V \not\subseteq Y\ or\ W \subseteq Y)\). We deduce that \(V \not\subseteq X \cap Y\ or\ W \subseteq X \cap Y\ and\ the\ result\ follows.\)

**Corollary 1.** Let \(Q\) be a SafeRL query and \(d\) a database. Then, \(Q\) and \(d\) define a unique closure system.

It turns out that for every RQL query, there exists a closure system and thus, the main problem is to effectively compute a representation of the closure system, i.e. a base (cf. definition 1).

In our setting, the definition of the base is:

**Definition 7.** Let \(Q = (\psi, \delta)\) be a SafeRL query over \(R\) and \(d\) a database over \(R\). We assume that the attribute variable in \(\delta\) is \(A\). The base of \(Q\) in \(d\), denoted by \(B_Q(d)\), is defined by:

\[
B_Q(d) = \bigcup_{\sigma, t \mid \langle d, \sigma \rangle \models \psi} \{A \in \text{sch}(Q) \mid \langle \sigma, \rho_{A \rightarrow t} \rangle \models \delta\}\]
We have to show that

\[ \forall \alpha \in \text{sch}(Q) \setminus \mathbb{Z}, \delta \mapsto \text{ans}(Q, \delta) \]

We know that \( \mathbb{Z} \in \mathbb{C} \mathbb{L}(Q(d)) \), we have to show that \( \mathbb{Z} \in B_Q(d) \). We know that \( \mathbb{Z} \in \mathbb{C} \mathbb{L}(Q(d)) \). Then, we have for all \( \alpha \in \text{sch}(Q) \setminus \mathbb{Z}, \mathbb{Z} \rightarrow \alpha \notin \text{ans}(Q, d) \). This means there exists some \( \alpha \) such that \( \langle \alpha, \rho_{A \rightarrow \alpha} \rangle \notin \delta(A, t_1, \ldots, t_n) \) and for all \( \beta \in \mathbb{Z}, \langle \alpha, \rho_{A \rightarrow \beta} \rangle \notin \delta(A, t_1, \ldots, t_n) \). In other words, for all \( \alpha \in \text{sch}(Q) \setminus \mathbb{Z}, \) there exists \( \mathbb{Z} \in B_Q(d) \) such that \( \mathbb{Z} \subseteq \mathbb{X} \) and \( \alpha \notin \mathbb{X} \). Since \( \mathbb{Z} = \bigcap_{\alpha \in \text{sch}(Q) \setminus \mathbb{Z}} \mathbb{X} \subseteq \mathbb{X} \) and \( \alpha \notin \mathbb{X} \) and \( \alpha \notin \mathbb{X} \) and \( \mathbb{Z} \in \mathbb{C} \mathbb{L}(Q(d)) \), we have the result, i.e. \( \mathbb{Z} \in B_Q(d) \).

#### 4.2. Computing the base of a SafeRQL query using query rewriting

The naive approach consists in executing \( n \) SQL queries against the database, caching all intermediary results, keeping only the right combination of tuples with respect to the WHERE clause and then computing the base of the closure system. We can do much better: the basic idea is to transform the query in order to push as much as possible the processing into the SQL query engine.

For every RQL query \( Q = \langle \psi, \delta \rangle \) involving \( n \) tuple-variables, there exists another query \( Q' = \langle \psi', \delta' \rangle \) with a unique tuple-variable. The practical consequence of this remark is that the computation of the base can be done in a single SQL query, i.e. the computation of the base of \( \langle \psi', \delta' \rangle \) can be delegated to the SQL query engine which was not the case in the former formulation.

By means of a rewriting function \( \text{rw} \), we have to transform \( Q = \langle \psi(t_1, \ldots, t_n), \delta(A, t_1, \ldots, t_n) \rangle \) into \( Q' = \langle \psi'(t), \delta'(A, t) \rangle \).

The idea of \( \text{rw} \) is that the unique tuple-variable \( t \) appearing in \( Q' \) takes its values in the schema built from the disjoint union of \( \text{sch}(t_i), i = 1..n \) and is essentially the concatenation of the initial \( t_i \)’s.

Let \( R \) be the new schema built as follows: \( R = \bigcup_{i = 1..n} \{ \langle A, i \rangle \mid \alpha \in \text{sch}(t_i) \} \). Let \( t \) be a fresh tuple variable with \( \text{sch}(t) = R \) and \( A_1, \ldots, A_n \) be \( n \) fresh attribute variables.

**Example 6.** Let us consider the relation \( \text{Empno} \) over attributes \( \text{Empno, Lastname, Workdept} \) shortened in the following examples to employees No. 10, 90 and 100. Let \( Q_1 = \langle \text{Empno}(t_1) \wedge \text{Empno}(t_2), (t_1.A = t_2.A) \rangle \).
Given $Q_1$, its rewriting is defined over the following new schema $\text{Emp}^*$:
$$\{ \langle \text{Empno}, 1 \rangle, \langle \text{Lastname}, 1 \rangle, \langle \text{Workdept}, 1 \rangle, \langle \text{Empno}, 2 \rangle, \langle \text{Lastname}, 2 \rangle, \langle \text{Workdept}, 2 \rangle \}$$. The rewriting will use a fresh variable $t : \text{Emp}^*$ and two fresh attribute variables $A_1$ and $A_2$.

Now, the rewriting function $\text{rw}$ has to be defined for TRC formulas, mining formulas and has to overload tuple variable assignments and attribute variable assignments.

For mining formulas, the rewriting is defined inductively. Let $\delta$ be a mining formula.

- $\text{rw}(t_i.A = c) = (t.A_i = c)$
- $\text{rw}(t_i.A = t_j.A) = (t.A_i = t.A_j)$
- $\text{rw}(\neg \delta) = \neg \text{rw}(\delta)$
- $\text{rw}(\delta_1 \land \delta_2) = \text{rw}(\delta_1) \land \text{rw}(\delta_2)$

Clearly, attribute variable names matter in this rewriting: $A_i$ carries both the name of the initial variable $A$ and the index $i$ of the initial tuple variable.

**Example 7.** In $Q_1$, let us consider the mining formula $\delta = (t_1.A = t_2.A)$. Then $\text{rw}(\delta) = (t.A_1 = t.A_2)$.

We now have to overload $\text{rw}$ to transform attribute-assignment $\rho$ and tuple-assignment $\sigma$ in order to take into account the new fresh variables.

- For each $A_i$ introduced above, we define $\text{rw}(\rho)(A_i) = (\rho(A), i)$, i.e. the name of the variable $A_i$ allows to identify both the attribute variable $A$ and the $i^{th}$ part of the tuple on the new schema $R$.
- Let $t : R$ the tuple variable introduced above. We define $\text{rw}(\sigma)(t) = \text{t}$ such that $\text{t}(\langle A, i \rangle) = \sigma(t_i)(A)$ for each $\langle A, i \rangle \in R$. Moreover, the tuple assigned to each $t_i$ by $\sigma$ is kept in $\text{rw}(\sigma)$: for $i \in \{1, \ldots, n\}$, $\text{rw}(\sigma)(t_i) = \sigma(t_i)$.

**Example 8.** Continuing the previous example with the following attribute-assignment $\rho_0(A) = \text{Lastname}$. We have:

- $\text{rw}(\rho_0)(A_1) = (\rho_0(A), 1) = \langle \text{Lastname}, 1 \rangle$
- $\text{rw}(\rho_0)(A_2) = (\rho_0(A), 2) = \langle \text{Lastname}, 2 \rangle$

Then, with the following tuple-assignment $\sigma_0(t_1) = t_1$, $\sigma_0(t_2) = t_2$, we have:

- $\text{rw}(\sigma_0)(t) = \text{t}$ such that $\forall A \in \{\text{Empno}, \text{Lastname}, \text{Workdept}\}$, $\text{t}(\langle A, 1 \rangle) = \sigma_0(t_1)(A) = t_1(A)$ and $\text{t}(\langle A, 2 \rangle) = \sigma_0(t_2)(A) = t_2(A)$. 

<table>
<thead>
<tr>
<th>Emp</th>
<th>Empno</th>
<th>Lastname</th>
<th>Workdept</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>10</td>
<td>SPEN</td>
<td>C01</td>
</tr>
<tr>
<td>$t_2$</td>
<td>90</td>
<td>HENDER</td>
<td>D21</td>
</tr>
<tr>
<td>$t_3$</td>
<td>100</td>
<td>SPEN</td>
<td>C01</td>
</tr>
</tbody>
</table>
In other words, the tuple \( t \) defined over \( \text{Emp}^* \) is equal to the concatenation of \( t_1 \) and \( t_2 \) in the original relation, namely \((t_1, \text{SPEN}, C01, 90, \text{HENDER}, D21)\) (cf. Figure 2 where \( t \) is represented by \( t'_1 \)).

Given a mining formula \( \delta \), we have the following lemma that gives an equivalence with the previous rewriting \( \text{rw}(\delta) \).

**Lemma 2.** \( \langle \sigma, \rho \rangle \models \delta \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \)

**Proof 3.** By induction on \( \delta \):

- If \( \delta = \neg \delta' \): \( \text{rw}(\delta) = \neg \text{rw}(\delta') \). By induction, \( \langle \sigma, \rho \rangle \not\models \delta' \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \not\models \text{rw}(\delta') \). Therefore \( \langle \sigma, \rho \rangle \models \delta \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \).

- If \( \delta = \delta_1 \land \delta_2 \): \( \text{rw}(\delta) = \text{rw}(\delta_1) \land \text{rw}(\delta_2) \). By induction, \( \langle \sigma, \rho \rangle \models \delta_1 \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta_1) \) and \( \langle \sigma, \rho \rangle \models \delta_2 \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta_2) \). Therefore \( \langle \sigma, \rho \rangle \models \delta_1 \land \delta_2 \) iff \( \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta_1) \land \text{rw}(\delta_2) \).

- If \( \delta = (t_i.A = t_j.A) \): \( \text{rw}(\delta) = (t_i.A = t_j.A) \). Let \( \sigma' = \text{rw}(\sigma) \) and \( \rho' = \text{rw}(\rho) \). By definition of \( \text{rw} \): \( \sigma(t_i)(\rho(A)) = \sigma'(t)(\rho(A)) \) and \( \sigma(t_j)(\rho(A)) = \sigma'(t)(\rho(A)) \).

  Let us assume that \( \langle \sigma, \rho \rangle \models \delta \). Then \( \sigma(t_i)(\rho(A)) = \sigma(t_j)(\rho(A)) \). Therefore \( \sigma'(t)(\rho'(A_i)) = \sigma'(t)(\rho'(A_j)) \) and \( \langle \sigma', \rho' \rangle \models t_i.A = t_j.A \).

  Conversely if \( \langle \sigma', \rho' \rangle \models \text{rw}(\delta) \), then \( \sigma'(t)(\rho'(A_i)) = \sigma'(t)(\rho'(A_j)) \). Therefore \( \sigma'(t)(\rho'(A_i)) = \sigma'(t)(\rho'(A_j)) \) and \( \langle \sigma, \rho \rangle \models t_i.A = t_j.A \).

- The case where \( \delta = (t_i.A = c) \) is similar to the previous one.

Finally, it remains to rewrite the TRC formula \( \psi(t_1, \ldots, t_n) \) with \( \text{rw}(\psi)(t) \) by forcing \( t \) to have each \( t_i \) as one of its components:

\[
\text{rw}(\psi) = \exists t_1 : \text{sch}(t_1) (\exists \ldots \exists t_n : \text{sch}(t_n) (\psi \land \bigwedge_{k \in \text{sch}(Q)} \bigwedge_{i=1}^n t.(A, i) = t_i(A)) \ldots )
\]

**Example 9.** Continuing the previous example, we have \( \psi(t_1, t_2) = \text{Emp}(t_1) \land \text{Emp}(t_2) \) and \( \text{sch}(Q_1) = \{ \text{Empno}, \text{Lastname}, \text{Workdept} \} \).

Then \( \text{rw}(\psi) = \exists t_1 : \text{sch}(t_1) (\exists t_2 : \text{sch}(t_2)(\zeta)) \) such that \( \zeta \) is the conjunction of the following conditions:

- \( \text{Emp}(t_1) \)
- \( \text{Emp}(t_2) \)
- \( t.(\text{Empno}, 1) = t_1.\text{Empno} \)
- \( t.(\text{Empno}, 2) = t_2.\text{Empno} \)
- \( t.(\text{Lastname}, 1) = t_1.\text{Lastname} \)
Proof 4. Let us assume that \( \langle d, \sigma \rangle = \psi \).

Since \( t \) is not a tuple-variable of \( \psi \), \( \langle d, \sigma \rangle = \psi \). By definition of \( \sigma \),
\( \langle d, \sigma \rangle \models [\sigma \in Q] \land_{i=1}^{n} t.\langle A, i \rangle = t_{i}.A \). Therefore \( \langle d, \sigma \rangle \models \exists t_{1} \ldots \exists t_{n}(\psi \land
\land_{i=1}^{n} t.\langle A, i \rangle = t_{i}.A) \). i.e. \( \langle d, \sigma \rangle = \psi \).

Let us now assume that \( \langle d, \sigma \rangle = \psi \) and consider the following formula: \( \psi(t_{1}, \ldots , t_{n}) \land_{x \in \text{sch}(Q)} \land_{i=1}^{n} t.\langle A, i \rangle = t_{i}.A \). Then there exists \( t_{1}, \ldots , t_{n} \)
and \( \sigma \) such that \( \sigma = \psi(t_{1}, \ldots , t_{n}) \land_{x \in \text{sch}(Q)} \land_{i=1}^{n} t.\langle A, i \rangle = t_{i}.A \). For \( \langle d, \sigma \rangle = \psi \land_{x \in \text{sch}(Q)} \land_{i=1}^{n} t.\langle A, i \rangle = t_{i}.A \) to hold, we necessarily have
\[ t_{i} = \sigma(t_{i}) \text{ for } 1 \leq i \leq n. \] Since \( \langle d, \sigma \rangle = \psi \), we deduce \( \langle d, \sigma \rangle = \psi \).

Example 10. Continuing the previous example, let us consider \( \psi(t_{1}, t_{2}) \), its
rewriting \( \psi(t_{1}, t_{2}) \), the associated query \( \{ t \mid \psi(t_{1}, t_{2}) \} \) and its evaluation against
\( \text{Emp} \). Figure 2 gives the result of this rewritten query. Note that, to avoid considering
each pair of employees twice, we have restricted this query to \( t_{1}.\text{Empno} < t_{2}.\text{Empno} \) i.e.
\[ Q_{1} = \langle \text{Emp}(t_{1}) \land \text{Emp}(t_{2}) \rangle \land t_{1}.\text{Empno} < t_{2}.\text{Empno}, (t_{1}.A = t_{2}.A) \).

Interestingly, for every \( \text{Safe} \text{RCL} \) query \( Q = \langle t_{1}, \ldots , t_{n} \rangle, \delta(t_{1}, \ldots , t_{n}, A) \),
\( \psi(t_{1}, t_{2}) \) gives an intermediate representation allowing to compute the mining formula
\( \delta(t_{1}, \ldots , t_{n}, A) \) on each tuple of the answer set.

Example 11. Let us consider the second tuple, denoted by \( t_{2}' \), given in example
10. The attribute \( \text{Lastname} \) (resp. \( \text{Workdept} \)) satisfies \( \psi(t_{2}') \) since \( t_{2}'(\langle \text{Lastname}, 2 \rangle) = t_{2}'(\langle \text{Lastname}, 1 \rangle) \) (resp. \( t_{2}'(\langle \text{Workdept}, 1 \rangle) = t_{2}'(\langle \text{Workdept}, 2 \rangle) \)). The
attribute \( \text{Empno} \) does not since \( t_{2}'(\langle \text{Empno}, 1 \rangle) \neq t_{2}'(\langle \text{Empno}, 2 \rangle) \). From \( t_{2}' \),
the maximal set of attributes verifying \( \psi(t_{2}') \) is \{ \text{Lastname}, \text{Workdept} \}, which
is a closed set associated to the initial query \( Q_{1} \). The closed set associated
with \( t_{2}'(\langle \text{A}, 1 \rangle) \) is empty since \( \forall A \in \text{sch}(Q_{1}), t_{2}'(\langle A, 1 \rangle) \neq t_{2}'(\langle A, 2 \rangle) \) (resp.
\( t_{2}'(\langle A, 1 \rangle) \neq t_{2}'(\langle A, 2 \rangle) \)).

We can now give an important result of the paper, i.e. how a base of the
closure system associated to every \( \text{Safe} \text{RCL} \) query can be computed.
Proposition 2.

\[ B_Q(d) = \bigcup_{\sigma \text{ s.t. } \langle d, \sigma \rangle \models \psi} \left\{ \{ A \in \text{sch}(Q) \mid \exists \rho : \rho(A) = A \land \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \} \right\} \]

Proof 5. By definition:

\[ B_Q(d) = \bigcup_{\sigma \text{ s.t. } \langle d, \sigma \rangle \models \psi} \left\{ \{ A \in \text{sch}(Q) \mid \exists \rho : \rho(A) = A \land \langle \sigma, \rho \rangle \models \delta \} \right\} \]

Since \( A \) is the only attribute-variable in \( \delta \):

\[ B_Q(d) = \bigcup_{\sigma \text{ s.t. } \langle d, \sigma \rangle \models \psi} \left\{ \{ A \in \text{sch}(Q) \mid \exists \rho : \rho(A) = A \land \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \} \right\} \]

By lemma 2:

\[ B_Q(d) = \bigcup_{\sigma \text{ s.t. } \langle d, \sigma \rangle \models \psi} \left\{ \{ A \in \text{sch}(Q) \mid \exists \rho : \rho(A) = A \land \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \} \right\} \]

By lemma 3:

\[ B_Q(d) = \bigcup_{\sigma \text{ s.t. } \langle d, \sigma \rangle \models \psi} \left\{ \{ A \in \text{sch}(Q) \mid \exists \rho : \rho(A) = A \land \langle \text{rw}(\sigma), \text{rw}(\rho) \rangle \models \text{rw}(\delta) \} \right\} \]

We conclude by remarking that \( \text{rw} \) is bijective for \( \sigma \).

The main theorem can now be given.

Theorem 1. Let \( Q \) be a \SafeRL query and \( d \) a database. Then

1. \( \text{ans}(Q,d) \) defines a unique closure system on \( \text{sch}(Q) \), denoted by \( \text{CL}(\text{ans}(Q,d)) \).

2. There exists an SQL query \( Q' \) over \( d \) such that \( \text{IRR}(\text{ans}(Q,d)) \subseteq \text{ans}(Q',d) \subseteq \text{CL}(\text{ans}(Q,d)) \).

Proof 6. The first item follows from Lemma 1. The second one follows from Proposition 1 and 2 for safe TRC formula. Since SQL is at least as expressive as safe TRC, the result holds.

Therefore \( B_Q(d) \) is computable by running only one SQL query, corresponding exactly to the safe TRC query \( \{ t \mid \text{rw}(\psi) \} \) with only one difference: the \texttt{SELECT} part has to evaluate the satisfaction of \( \text{rw}(\delta) \). From a practical point of view, it means that the base can be computed with only one SQL statement for all \SafeRL queries, pushing query processing as much as possible into the DBMS.
5. Implementation and experiments

Given a RQL query Q, the query processing engine consists of a Java/JavaCC application to:

1. Compute the base of the closure system of Q using the generated SQL query provided by Theorem 1.
2. From the base, compute the canonical cover for exact rules and a Gottlob and Libkin cover for approximate rules [29]. Details are out of the scope of this paper, we mainly reused the code of T. Uno [45] for the most expensive part of the rule generation process, i.e. the enumeration of minimal transversal of hypergraphs.

Figure 3 gives an overview of RQL query processing.

5.1. Sensor Data

We experimented our RQL processing engine using the PlaceLab dataset provided by the MIT House Consortium and Microsoft Research [34].

The PlaceLab is a 93 m² apartment instrumented with several hundred sensors. During the experiment, interior conditions (temperature, humidity, light, pressure, current, gas and water flow) were captured by 106 sensors, along with 92 reed switches installed on doors, drawers and windows to detect open-closed events. 277 wireless motion sensors were placed on nearly all objects that might be manipulated. Two participants used the PlaceLab as a temporary home for 10 weeks.

The available data is a subset of about a month from the original 2.5 months experiment, from August 22, 2006 to September 18. Raw data is extracted from binary files, where each reading contains a sensor id, a timestamp and a value (the measurement). Sensors meta-data include, for each sensor id, type, location and a short textual description, along with other meta-data of little interest for our experiments, such as installation date, etc. This data is stored in an Oracle database whose schema is depicted in figure 4.
samples

<table>
<thead>
<tr>
<th>id</th>
<th>DECIMAL(20,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>type</td>
<td>DECIMAL(3,0)</td>
</tr>
<tr>
<td>value</td>
<td>DECIMAL(10,0)</td>
</tr>
</tbody>
</table>

descriptions

<table>
<thead>
<tr>
<th>id</th>
<th>DECIMAL(20,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>VARCHAR(12)</td>
</tr>
<tr>
<td>location</td>
<td>VARCHAR(18)</td>
</tr>
<tr>
<td>description</td>
<td>VARCHAR(78)</td>
</tr>
</tbody>
</table>

Figure 4: PlaceLab database schema

<table>
<thead>
<tr>
<th>time</th>
<th>bathroom_light</th>
<th>kitchen_humidity_0</th>
<th>...</th>
<th>bedroom_temperature_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-08-22 00:00:00</td>
<td>0.4971428</td>
<td>4344</td>
<td>...</td>
<td>21.43</td>
</tr>
<tr>
<td>2006-08-22 00:01:00</td>
<td>0.6685879</td>
<td>4344</td>
<td>...</td>
<td>21.43</td>
</tr>
<tr>
<td>2006-08-22 00:02:00</td>
<td>0.4985673</td>
<td>4344</td>
<td>...</td>
<td>21.465</td>
</tr>
<tr>
<td>2006-09-18 23:58:00</td>
<td>1567.7822</td>
<td>5324</td>
<td>...</td>
<td>22.53</td>
</tr>
<tr>
<td>2006-09-18 23:59:00</td>
<td>1563.5891</td>
<td>5276</td>
<td>...</td>
<td>22.50</td>
</tr>
</tbody>
</table>

Figure 5: Sensors data after SQL preprocessing

For data mining queries, we focused on variations of the physical properties of the environment, such as temperature, light, etc., which amount to more than 100 million tuples. A view, easily expressed with SQL, has been created to synchronize and resample every sensor with a common sampling interval (one minute). This view, illustrated in figure 5, has 165 attributes and 32543 tuples. Apart from the time, each attribute is associated either with one of the 106 physical properties sensors or one of the 58 selected switches.

5.2. Experimental Results

The server used for these experiments is a virtual machine running on VMware 4.1 with 2× Intel Xeon X5650 and 7.2K disks in RAID 5. This virtual machine, running Debian 6.0, disposes of 16 GB of RAM and 4 CPU cores. The DBMS is Oracle Database 11g Release 2.

In these experiments, we consider three families of RQL queries.

Null values (Q₃). This first set of queries mine rules between sensors for null values. Such queries can be used to help identify groups of sensors which are unavailable simultaneously, due, for example, to a shared acquisition system.

Q₃: FINDRULES

OVER <list of attributes>
SCOPE t1 sensors
CONDITION ON $A IS t1.$A IS NULL

For example, if we consider all temperature sensors as the list of attributes, we can group sensors dining_room_temperature_1 (A), dining_room_temperature_2 (B), dining_room_temperature_3 (C), dining_room_temperature_4 (D), hall_temperature_0 (E) and hall_temperature_3 (F) according to rules (A → F, F → C, BC → E, BE → D, CD → E, CE → A). Note that, following Armstrong’s axioms, rules such as BC → ADEF can be inferred. Using SQL queries, we can then confirm that these sensors are indeed almost always unavailable simultaneously. Similarly, we can group four sensors from the living
This information could especially be useful if we had lost metadata on sensors locations and descriptions.

Figure 6 gives the cumulative execution time for various number of attributes in the OVER clause of Q1. As expected, rule generation is by far the most expensive part when the query runs over a large set of attributes. The SQL query however lasts less than a second and increases linearly: computation of the base by the DBMS is efficient.

For the subsequent experiments, we focus on the execution time of the base computation. As seen in section 2.4, the data analyst does not always need to generate a canonical cover of the rules, and can address other problems directly from the base.

Functional dependencies (Q4). This second set of queries finds FDs within a time window. This time window is specified using SQL conditions on the timestamp.

\( Q4: \text{FINDRULES} \)

\[
\text{OVER <list of attributes>}
\]

\[
\text{SCOPE t1, t2 (}
    \text{SELECT * FROM sensors}
    \text{WHERE time BETWEEN '2006-08-22 00:00:00'
    AND <end date>}
\]

\[
\text{WHERE t1.rowid < t2.rowid}
\]

\[
\text{CONDITION ON $A IS t1.$A = t2.$A}
\]

To generate the base, the corresponding SQL query performs a Cartesian product (more precisely, a theta self-join on \( t1.rowid < t2.rowid \), which gives half as many tuples). Consequently, the SQL part is significantly more costly compared with the previous family of RQL queries. Figure 7 shows the scalability of the base generation with respect to the number of attributes (\(#\text{tuples} = 1439\) and the number of tuples (\(#\text{attributes} = 37\) ). As expected, the execution time
increases quadratically with the number of tuples due to the theta self-join.

Local maximums ($Q_5$). RQL queries can express a wide range of conditions for rules. For example, the following query finds rules for local maximums of measurements (i.e. $X \rightarrow A$ is interpreted as: if all sensors in $X$ have a local maximum at time $T$, then sensor $A$ has a local maximum at time $T$).

$Q_5$:

```
FINDRULES
OVER <list of attributes>
SCOPE t1, t2, t3 sensors
WHERE t2.time = t1.time + interval '1' minute
AND t3.time = t2.time + interval '1' minute
CONDITION ON $A IS t1.$A < t2.$A AND t2.$A > t3.$A
```

Even though this query has three tuple-variables, all three are bound by a 1:1 relationship. Consequently, the DBMS computes equi-joins instead of Cartesian products, which improves its efficiency. Figure 8 shows that, similar to $Q_3$, the execution time grows linearly with both the number of attributes ($\#tuples = 32433$) and the number of tuples ($\#attributes = 37$) – a time window is added to the SCOPE clause to adjust the number of tuples, cf. $Q_4$.

5.3. Comparison with functional dependencies miners

Due to its general application case, RQL can be compared with existing tools for several data mining tasks. In this section, we focus on functional dependencies to evaluate how well RQL behaves with respect to dedicated solutions.

We compared RQL with two functional dependency discovery algorithms: TANE [32] and FDEP [23], identified to be among the fastest existing algorithms in a recent survey [51]. The implementations used as a basis for this comparison are taken from the Metanome project [50], along with five datasets: ncvoter, fdreduced30, plista, flight and uniprot. Other datasets used in this experiment are provided by the UCI machine learning repository [36]. Table 2 gives an overview of these datasets and the corresponding execution times for RQL,
FDEP and TANE, which includes: data retrieval from the Oracle database, FD discovery and writing the cover of minimal FDs to an output file.

The execution strategy of RQL is similar to FDEP, where a negative cover of maximal non-FDs is computed first. During this step, both algorithms have to compare records pair-wise, which is integrated in FDEP and delegated to the DBMS by RQL. Their behaviors are therefore comparable, with a good scalability on the number of columns but limited on the number of rows.

TANE implements another discovery strategy based on a bottom-up traversal of a lattice representing all possible subsets of attributes, along with a compact representation of the data (called partitions) used to test functional dependencies. In this experiment, TANE is the most efficient strategy on datasets with many rows, but exceeds the memory limit when its lattice becomes too large [51].

Despite not being dedicated to FDs discovery, RQL provides, in some cases (e.g. when the number of generated FDs is large), the most efficient strategy.

6. A web prototype for RQL

A web prototype has been developed and is freely accessible at http://rql.insa-lyon.fr for research and educational purposes. Figure 9 gives a preview of this web interface, which provides an unified access to the user’s data and pattern mining techniques using two declarative languages: SQL and RQL.

To discover the RQL language, a pre-filled database is made available in a Sample mode, with a few selected example written in both SQL and RQL. Data analysts can also upload their own data (up to 200 KB) in a Sandbox mode and write their own SQL and RQL queries to discover meaningful rules.

For example a data analyst wants to explore functional dependencies using $Q_1$. The analyst could generate a canonical cover for FDs, but instead she only wants to investigate if an employee can be identified by its name and department ($Lastname, Workdept \rightarrow Empno$). RQL answers whether or not this rule holds,
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Columns (#)</th>
<th>Rows (#)</th>
<th>Size (KB)</th>
<th>FDs (#)</th>
<th>RQL</th>
<th>FDEP</th>
<th>TANE</th>
</tr>
</thead>
<tbody>
<tr>
<td>iris</td>
<td>5</td>
<td>150</td>
<td>4</td>
<td>4</td>
<td></td>
<td>&lt;0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>balance-scale</td>
<td>5</td>
<td>625</td>
<td>6</td>
<td>1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>chess</td>
<td>7</td>
<td>28,056</td>
<td>519</td>
<td>1</td>
<td>214.6</td>
<td>71.6</td>
<td>1.0</td>
</tr>
<tr>
<td>abalone</td>
<td>9</td>
<td>4,177</td>
<td>187</td>
<td>137</td>
<td>5.9</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>nursery</td>
<td>9</td>
<td>12,960</td>
<td>1,036</td>
<td>1</td>
<td>60.9</td>
<td>25.1</td>
<td>6.4</td>
</tr>
<tr>
<td>breast-cancer</td>
<td>11</td>
<td>699</td>
<td>19</td>
<td>46</td>
<td>0.4</td>
<td>0.5</td>
<td>0.9</td>
</tr>
<tr>
<td>bridges</td>
<td>13</td>
<td>108</td>
<td>6</td>
<td>142</td>
<td>0.2</td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>echocardiogram</td>
<td>13</td>
<td>132</td>
<td>6</td>
<td>536</td>
<td>0.1</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>adult</td>
<td>15</td>
<td>48,842</td>
<td>3,881</td>
<td>66</td>
<td>1661.7</td>
<td>784.3</td>
<td>127.1</td>
</tr>
<tr>
<td>letter</td>
<td>17</td>
<td>20,000</td>
<td>696</td>
<td>61</td>
<td>327.7</td>
<td>158.0</td>
<td></td>
</tr>
<tr>
<td>ncvoter</td>
<td>19</td>
<td>1,000</td>
<td>151</td>
<td>758</td>
<td>1.1</td>
<td>1.0</td>
<td>1.7</td>
</tr>
<tr>
<td>hepatitis</td>
<td>20</td>
<td>155</td>
<td>7</td>
<td>8,250</td>
<td>0.6</td>
<td>0.5</td>
<td>5.2</td>
</tr>
<tr>
<td>horse</td>
<td>28</td>
<td>368</td>
<td>28</td>
<td>156,723</td>
<td>2.9</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>fdreduced30</td>
<td>30</td>
<td>250,000</td>
<td>69,580</td>
<td>89,571</td>
<td>TL</td>
<td>ML</td>
<td>132.9</td>
</tr>
<tr>
<td>plista</td>
<td>63</td>
<td>1,001</td>
<td>575</td>
<td>178,152</td>
<td>11.2</td>
<td>16.4</td>
<td></td>
</tr>
<tr>
<td>flight</td>
<td>109</td>
<td>1,000</td>
<td>569</td>
<td>982,631</td>
<td>9.3</td>
<td>139.3</td>
<td></td>
</tr>
<tr>
<td>uniprot</td>
<td>223</td>
<td>1,000</td>
<td>2439</td>
<td>unknown</td>
<td>TL</td>
<td>ML</td>
<td></td>
</tr>
</tbody>
</table>

**TL:** time limit of 4 hours exceeded  
**ML:** memory limit of 10 GB exceeded

Table 2: Execution times in seconds for functional dependency discovery

---

**Figure 9: Main web interface**
Rule verification:

The rule Lastname Workdept Empno is false

Counter-example:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>SPEN</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>JOB</th>
<th>EDUCLEVEL</th>
<th>SEX</th>
<th>SAL</th>
<th>BONUS</th>
<th>COMM</th>
<th>MGRNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>SPEN</td>
<td>C01</td>
<td>FINANCE</td>
<td>18</td>
<td>M</td>
<td>26150</td>
<td>20150</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SPEN</td>
<td>C01</td>
<td>FINANCE</td>
<td>18</td>
<td>F</td>
<td>52750</td>
<td>52750</td>
<td>4220</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Generated query:

1. SELECT t1.*, t2.*
2. FROM Emp t1, Emp t2
3. WHERE (t1.Lastname = t2.Lastname AND t1.Workdept = t2.Workdept)
4. AND CASE WHEN (t1.Empno = t2.Empno) THEN 1 ELSE 0 END = 0
5. AND rownum <= 1

Figure 10: Counter-examples with RQL

and if not, a counter example from the database is displayed. Figure 10 gives an overview of what RQL provides as a counterexample for this rule.

Importantly, the web interface provides the SQL code generated by the system as often as possible, for instance to identify the counterexample, so that the analyst can use this query as a starting point for data exploration [18].

7. Conclusion

In this paper, we have introduced SafeRL, a logical query language based on the tuple relational calculus and devoted to rule discovery in databases. The rule mining problem is seen as a query processing problem, for which we have proposed a query rewriting technique allowing the delegation of as much processing as possible to the underlying DBMS engine. RQL, the concrete language of SafeRL, is an SQL-like rule mining language which allows SQL developers to extract precise information without specific knowledge in data mining. A system has been developed and tested against a real-life database provided by the MIT House-n Consortium and Microsoft Research. These experiments show both the feasibility and the efficiency of the proposed language.

RQL is an important contribution toward declarative pattern mining whose ambition is to simplify the way data analysts interact with their data. Through a convenient query language, close to SQL, they can focus on scientific discovery instead of spending considerable time switching data between different systems.

As future work, we could generate other covers for rules with RQL, typically optimal covers, namely the Duquenne and Guigues basis [30] or ordered direct implicational basis [4] (see [2] for a recent survey). The scalability of RQL to Big Data remains a quite challenging issue. Even if there is still room for improvements, we believe that alternative approaches should be conducted to guide the search towards interesting rules without enumerating all possible satisfied rules, for example by taking advantage of the knowledge brought by counter-examples. From an application point of view, the potential of RQL could also be studied to express temporal dependencies [53] and more generally for data cleaning [28, 21].

25
Acknowledgment. This work has been partially funded by the French national research agency (DAG project, ANR-09-EMER, 2009-2013) and CNRS Mastodons projects (PETASKY 2012-2015 and QualiSky 2016).

References


