Querying Multiple Sets of Discovered Rules

Alexander Tuzhilin
IOMS Department
Leonard N. Stern School of Business
New York University
New York, NY 10012 USA
atuzhili@stern.nyu.edu

Bing Liu*
School of Computing
National University of Singapore
3 Science Drive 2
Singapore 117543
liub@comp.nus.edu.sg

ABSTRACT

Rule mining is an important data mining task that has been applied to numerous real-world applications. Often a rule mining system generates a large number of rules and only a small subset of them is really useful in applications. Although there exist some systems allowing the user to query the discovered rules, they are less suitable for complex ad hoc querying of multiple data mining rulebases to retrieve interesting rules. In this paper, we propose a new powerful rule query language Rule-QL for querying multiple rulebases that is modeled after SQL and has rigorous theoretical foundations of a rule-based calculus. In particular, we first propose a rule-based calculus ***RC*** based on the first-order logic, and then present the language Rule-QL that is at least as expressive as the safe fragment of ***RC***. We also propose a number of efficient query evaluation techniques for Rule-QL and test them experimentally on some representative queries to demonstrate the feasibility of Rule-QL.

Keywords

Data mining queries, rulebases, association rules, query languages, query evaluation.

1. INTRODUCTION

Rule mining is one of the central tasks of data mining. Rules can be utilized to gain a better understanding of the application domain, and to take actions to one's advantage. However, the number of rules generated from a dataset by some of the rule mining algorithms is often very large, in thousands or tens of thousands [4,13,15,16,17,20,21,24]. Making sense of such a large number of rules presents a significant challenge. Past research has shown that most of the rules are actually not useful or interesting for specific applications [15,16,17,20,21,24].

To help the user find interesting rules from a set of discovered rules, several methods have been proposed. One of them is *data mining queries* [10,14,15,18,23,28], which allow the user to focus only on some subsets of the rules that are of interest to him/her by asking appropriate queries. Although there have been several proposals reported in the literature on how to define data mining queries, all of them have one common basis: they use first-order logic to let the user specify what kinds of rules the querying system should retrieve from the rulebase of discovered rules.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGKDD '02, July 23-26, 2002, Edmonton, Alberta, Canada.
Copyright 2002 ACM 1-58113-567-X/02/0007…$5.00.

However, as will be discussed in Section 2, all authors focus on querying a *single* rulebase at a time and therefore restrict their query languages to particular fragments of first-order logic. It is important to expand previously proposed data mining query languages to the *full* set of first-order logic expressions operating on *multiple* rulebases because many practically important data mining queries naturally require multiple rulebases or at least a join of a rulebase with itself. Some examples of such queries are:

- In a rulebase of newly discovered rules, find exceptions (unexpected refinements [20,25]) for the set of previously discovered rules (stored in a separate rulebase).
- Find minimal rules in a rulebase. A minimal rule is a most general rule: its left-hand-side (LHS) and right-hand-side (RHS) do not contain the left-hand-side and the right-hand-side of any other rule in the rulebase respectively. Although this query operates on a single rulebase, it requires a join of the rulebase to itself, as will be shown later.
- Find rules that were stable over the last 6 months, assuming that the mined rules for each month are stored in a separate rulebase. This type of queries is important as data mining is increasingly used in the production mode. In such situations, the user often wants to see the changing behavior of the rules over a number of time periods [17]. Stable rules are those rules whose confidences and/or supports do not change significantly over time. Similarly, it is also useful to find rules whose confidences and/or supports are "growing" or "diminishing" over time, as they may indicate trends.

In this paper, we propose a more powerful rule query language Rule-QL that allows the user to query multiple rulebases. Each rulebase typically contains rules mined from a different dataset. Rule-QL is complete in the sense that it is based on the full set of first-order logic (FOL) expressions. To ground Rule-QL in a sound theory, we first introduce a logic-based calculus ***RC*** defined over a *set* of rulebases of association rules[1] and also define *safety* of ***RC*** expressions. Then we introduce Rule-QL that is at least as expressive as the safe fragment of ***RC***. We also introduce additional constructs in Rule-QL to increase its expressive power and make it more useful as SQL is more expressive than relational calculi. We also study processing of Rule-QL queries and present efficient query processing methods. We tested these methods on a number of typical queries and report the performance results that clearly indicate the feasibility of Rule-QL.

* Bing Liu is now with Department of Computer Science, University of Illinois at Chicago, liub@cs.uic.edu.

[1] Although we limit our approach to association rules, it is actually applicable to broader classes of rules. However, this would require introduction of additional constructs that go beyond the scope of this paper.

Previously proposed data mining query languages made the discovered rules more accessible to the end-users. We believe that Rule-QL and its rule management system will enhance this process further by allowing the users querying multiple rulebases in a more general and flexible manner. This should also make data mining results more useful in practice.

In summary, the contributions of this paper are:

- Development of a rule query language Rule-QL that is based on a theoretically sound rule-based calculus ***RC*** supporting queries on multiple rulebases.
- Presentation of efficient query processing methods for Rule-QL.
- Empirical evaluation of these query processing methods and presentation of performance results.

2. RELATED WORK

Facilitating the user to retrieve interesting rules has been studied by a number of researchers within the context of data mining query languages. One of the earliest data mining query language is the one based on templates [15]. In this technique, the user uses a template to specify what items should be in or not in a rule, and what level of support and/or confidence are required. The system then checks each mined rule to find those rules matching the template. This template-based language can be seen as a special case of our proposed language, Rule-QL. Our query language is much more powerful and expressive because it operates on several rulebases and supports a full set of first-order logic expressions defined on these rulebases.

In [10], Han *et al* presented a data mining query language, called DMQL. DMQL allows the user to specify from what table (and database) to mine what types of rules. Its main purpose is to select the right data to mine different types of rules. It is not a language for querying the discovered rules, which is the aim of our proposed language. Rule-QL is only used after rule mining has been completed. Thus, it is complementary to DMQL, where DMQL can be used to choose data to mine rules. After the rules are discovered, Rule-QL can be used by the user to retrieve various rules that are of interest to him/her. [18] proposes an SQL-like operator for data mining (MINE RULE). Also, [23] reports a metaquery language for data mining. Both these approaches are similar to DMQL. They are not designed for the purpose of querying the mined rules as Rule-QL, but enabling the user to specify the data mining task to perform and its required data. In addition, [11] describes different types of constraints that the user can provide for focusing the discovery process.

[14,28] report a more powerful data mining query language, called MSQL. MSQL can be used not only for rule generation, which is similar in capability to DMQL and the MINE RULE operator, but also for querying the discovered rules. With regard to rule querying, MSQL is similar to templates [15] but allows more complex conditions. A rule query can have three main types of conditions. The first type is the *rule format conditions*, which enables the user to express the requirements that a rule's LHS/RHS (left-hand-side/right-hand-side) must contain certain items, be a subset of a set of given items, or be equal to a set of given items. The second type is the *pruning conditions*, which express requirements that a rule must satisfy some constraints on confidence, support, or rule length. The third type is the *mutex conditions*, which specify that if a rule contains a particular given item, it should not contain any item of a set of other given items. Like templates, MSQL's query conditions can be checked using the information contained in each rule itself, e.g., support, confidence, rule length, items in LHS/RHS. That is, condition checking on each rule is independent of other rules. Furthermore, it can only operate on one rulebase at a time. In contrast to MSQL, Rule-QL operates on multiple rulebases at a time and can also support queries expressing inter-relationships of rules.

In [3], Agrawal *et al* reported a language for querying shapes of history. The shapes of history refer to ups and downs of supports or confidences of a rule over a number of time periods. This language enables one to define the behavior shapes of the rules that s/he is interested in. For example, one may want to find rules whose support increases in one time period and then falls down. This is different from our work, as [3] presents not a general query language for rules, but only for shapes. It is related to our work as we also allow querying rulebases mined from different time periods. Moreover, we introduce statistical procedures to find various statistically interesting rules. Only using ups and downs to express the change of a rule lacks statistical foundation.

[19] proposes an index for retrieval of association rules stored in a relational database. The purpose is to retrieve rules containing a given subset of items in their LHS or RHS. Rule-QL is more expressive and more efficient as rules can be represented more naturally as sets (rather than relational tables), and a more suitable index of inverted lists can be used to access them.

Another related research is the subjective and objective interestingness of data mining rules [4,12,16,17,20,21,24,25,29]. In subjective interestingness, [16,20,21,24,25] propose several methods for finding unexpected rules. These approaches first ask the user to specify his/her existing knowledge about the domain. The system then finds those unexpected rules by comparing the user's knowledge with the discovered rules. These methods are different from our query language. In fact, we will show that the task of finding unexpected rules can be formulated just as a query in Rule-QL. For objective interestingness, [12,26] present and review a number of methods for ranking rules according to different measures. Such operations can also be done in Rule-QL by issuing appropriate queries.

In summary, there have been several excellent proposals made on how to query rulebases. However, nobody developed a data mining query language on *multiple* rulebases that supports a full set of first-order logic expressions.

3. THE RULE-BASED CALCULUS *RC*

In this section we introduce a rule-based calculus ***RC*** that provides a theoretical basis for the rule query language Rule-QL presented in this paper.

Let $R_1, R_2, \ldots, R_n$ be rulebases (which are also called *rule sets*). To be specific, we assume that each rulebase contains association rules defined on some set of transactions T over the itemset I [2]. However, our approach is not limited to association rules and can also be applied to a broader set of rules. The vocabulary of the *rule-based calculus* ***RC*** is defined over

- Rulebases $R_1, R_2, \ldots, R_n$.
- A set of constants from the itemset I and a set of real numbers (to define restrictions on confidence and support).
- Rule-based variables $r_1, r_2, \ldots$ (also called rule variables) defined over rulebases $R_1, R_2, \ldots, R_n$.

- Functions LHS and RHS that map association rules into itemsets obtaining the left- and the right-hand-sides of a rule respectively; functions CONF and SUP that map association rules into reals and specify the confidence and support of a rule respectively.
- Relational operators $\subset, \subseteq, \supset, \supseteq$, and = defined on itemsets in a standard manner, and relational operators ≥, ≤, <, >, = defined on real numbers (for confidence and support comparisons).

An *atomic formula* of ***RC*** has one of the following forms:

- $R(r)$, where R is a rulebase and r a rule-based variable.
- α *op* β, or α *op const,* where α and β are functions LHS(r) or RHS(r), *const* is a set of items (e.g. {milk, bread}) and *op* is one of the operators $\subset, \subseteq, \supset, \supseteq$ or =.
- γ *op* δ or γ *op const,* where γ and δ are functions CONF(r) or SUP(r), *const* is a real constant, and *op* is one of the standard relational operators ≤, ≥, >, <, =.

A set of well-formed formulae in ***RC*** is obtained from atomic formulae in a standard way as it is done in first-order logic by taking the closure of conjunction, disjunction, negation and universal and existential quantification operators. *Safety* of ***RC*** formulae is defined similarly to the safety of relational calculus formulae [1]. More specifically, we first define a *safe-range normal form* (SRNF) in the same way as [1] does. Then, following [1], we define the set of *range-restricted variables* of an SRNF formula in a very similar way with certain modifications pertaining to rulebases and rules. In particular, range restricted variables of a ***RC*** formula φ are defined recursively as $rr(\varphi)$ =

1. r, if φ is $R(r)$, where R is a rulebase
2. r, if φ has LHS(r) = $const_1$ and RHS(r) = $const_2$ where $const_1$ and $const_2$ are sets of items
3. $rr(\psi) \cup r$, if $\varphi \equiv \psi \wedge$ RHS(r) = L|RHS(s) $\wedge$ LHS(r) = L|RHS(t)[2] and $s \in rr(\psi) \wedge t \in rr(\psi)$;
   $rr(\psi)$ otherwise.
4. $rr(\varphi_1) \cup rr(\varphi_2)$, if $\varphi = \varphi_1 \wedge \varphi_2$
5. $rr(\varphi_1) \cap rr(\varphi_2)$, if $\varphi = \varphi_1 \vee \varphi_2$
6. $\varnothing$, if φ is $\neg\psi$.
7. $rr(\psi) - r$ if $\varphi \equiv (\exists r)\, \psi$ and $r \in rr(\psi)$;
   $\perp$ (undefined), if $\varphi \equiv (\exists r)\, \psi$ and $r \notin rr(\psi)$.

A ***RC*** formula φ is *safe* if rr(SRNF(φ)) is equal to the set of all the free variables of φ.Finally, a ***RC*** *query* is defined as $\{r \mid \varphi\}$, where φ is a safe ***RC*** formula having only *one* free rule-based variable r.

The semantics of formula φ is defined in usual model-theoretic terms, where $R_1, R_2, ..., R_n$ are interpreted as sets of (association) rules, and binary relational operators $\subset, \subseteq, \supset, \supseteq$, ≥, ≤, <, > and =, and functions LHS, RHS, CONF and SUP are interpreted in the standard way. The semantics of query $\{r \mid \varphi\}$ is also defined as in the relational case, where we use rulebases instead of databases.

Throughout this section, we emphasize similarities between the rule-based calculus ***RC*** and the classical tuple relational calculus. However, there are significant differences between the two languages. First of all, ***RC*** deals with rules and rulebases that constitute totally different concepts from relations. Secondly, ***RC*** supports several rule-specific constructs, such as functions LHS, RHS, CONF, SUP, and containment operators on itemsets, such as $\subset, \subseteq, \supset, \supseteq$, etc.

[2] The notation L|RHS denotes "either LHS or RHS."

Finally, as was pointed out in Section 2, ***RC*** significantly differs from previously proposed data mining query languages in that it supports multiple rulebases and the full set of FOL expressions over these rulebases.

We would also like to point out that we introduced only the most basic constructs in ***RC*** trying to keep the language simple. Clearly, we could also add additional operators to the calculus, making it more expressive. For example, we could add functions SIZE_OF specifying the length of a rule, arithmetic operators over real numbers (in order to do arithmetic over confidences and supports of rules) and union and intersection operations over itemsets (and thus be able to express additional queries in ***RC***). Instead of defining these constructs, we decided to keep the language ***RC*** simple and introduce some of these extra constructs in the query language Rule-QL that is based on ***RC.***

We next present some examples of ***RC*** queries. We assume that the rules are defined over the standard market basket rulebase (MB) over the items of groceries that is typically considered in the literature on association rules.

Query 1: *Find all the association rules referring to milk and cereal in the body (or LHS) and having confidence over 90%.*

$$\{r \mid MB(r) \wedge LHS(r) \supseteq \{milk, cereal\} \wedge CONF(r) > 90\%\}$$

This simple query can easily be expressed in the languages of [15,28]. However, the rest of the queries presented in this section cannot be expressed in these languages because they either operate on multiple rulebases or require quantification over some other rule-based variables, as is the case with the following query.

Query 2: *Find minimal association rules, i.e., the rules whose LHS and RHS cannot be reduced further.*

$$\{r \mid MB(r) \wedge \neg(\exists r')(MB(r') \wedge r \neq r' \wedge LHS(r') \subseteq LHR(r) \wedge RHS(r') \subseteq RHS(r))\}$$

Query 3: *Find all the rules from rulebase NEWRULES that are exceptions of some of the previously discovered rules OLDRULES.*

$$\{r \mid NEWRULES(r) \wedge (\exists r')\,(OLDRULES(r') \wedge LHS(r') \subset LHS(r) \wedge RHS(r') \neq RHS(r))\}$$

Query 4: *Find all the rules that were held over the last 3 months with confidence at least 60% (we assume that a separate rulebase MONTHi exists for each month i).*

$$\{r \mid (\exists r', r'')\,(MONTH1(r) \wedge MONTH2(r') \wedge MONTH3(r'') \wedge LHS(r) = LHS(r') = LHS(r'') \wedge RHS(r) = RHS(r') = RHS(r'') \wedge CONF(r) > 60\% \wedge CONF(r') > 60\% \wedge CONF(r'') > 60\%)\}$$

One of the major features of calculus ***RC*** is that its queries are generally defined over *multiple* rulebases, as we can see in Queries 3 and 4. Therefore, ***RC*** is more expressive than previously introduced data mining query languages, such as templates and MSQL, that operate on a single rulebase.

4. RULE QUERY LANGUAGE Rule-QL

In this section, we take calculus ***RC*** as a theoretical foundation for the data mining query language Rule-QL. In particular, we introduce a SQL-like syntax that can express all ***RC*** queries and also add several additional operators and functions to the language in order to make Rule-QL more expressive and versatile. Some examples of these constructs are functions SIZE_OF, MAX, MIN, AVG and COUNT, clauses GROUP BY, HAVING and ORDER, and the operator UNION. In our presentation of Rule-QL, we start with a basic version of the language and then introduce more advanced features. The structure of a basic Rule-QL query is

```
SELECT [FUNCT]r
FROM   R r [, Ri ri]*
WHERE  Conditional_Expression
```

The FROM clause above specifies one or several rulebases R, R_1, …, R_n over which Rule-QL queries are defined. Also, r, r_1, …, r_n are the rule variables defined over these rulebases. The SELECT clause selects *one* variable (over its corresponding rulebase R) that satisfies the conditions of the WHERE clause. In addition, optional function FUNCT specifies various aggregation operations over the resulting set of rules. Some examples of function FUNCT are

- COUNT: counts the number of rules in the answer to a query.
- AVG_CONF, AVG_SUP, AVG_SIZE: they determine the average levels of confidence, support and the size of the rules in the answer respectively.
- MAX_SUP, MAX_CONF, MAX_SIZE, MIN_SUP, MIN_CONF, MIN_SIZE: they are defined similarly to AVG_CONF, AVG_SUP, AVG_SIZE, but deal with maximal and minimal values rather than averages.

Finally, the *Conditional_Expression* in the WHERE clause is defined as follows. A *simple* condition is defined as

- L|RHS(*r*) *op* L|RHS(*s*) or L|RHS(*r*) *op const*, where L|RHS(r) is either the LHR or the RHS of rule *r*, *op* is one of the standard relational operators CONTAINED_IN, CONTAINED_EQ_IN, CONTAINS, CONTAINS_EQ and EQUAL corresponding to the ***RC*** operators $\subset, \subseteq, \supset, \supseteq$ and = respectively, and *const* is an itemset.
- *Rule_funct*(*r*) *rel_op const* or *Rule_funct*(*r*) *rel_op Rule_funct*(*s*), where *Rule_funct* is one of the functions defined on a rule, such as CONF, SUP and SIZE_OF defining confidence, support, and the size of a rule (the number of items in the rule) respectively; also *rel_op* is one of the relational operators $\leq, \geq, >, <, =$, and *const* is a constant.
- *Item_funct*(*I*) *rel_op const* or *Item_funct*(*I*) *rel_op Item_funct*(*I'*), where *Item_funct* is one of the functions defined on a set of items, such as function SIZE_OF determining the size of an itemset.
- EXISTS (<*Rule_Query*>), where *Rule_Query* is an arbitrary Rule-QL query.
- *r* [*NOT*] IN <*Rule_Query*>, where *r* is a rule and *Rule_Query* is as specified above.

A complex *Conditional_Expression* is defined in terms of simple conditional expressions in the same way as it is done in SQL as a disjunction of conjunctive clauses that also support negations [5].

We next give some examples of the basic Rule-QL queries, starting with the queries introduced in Section 3.

Query 1. Simple selection with a single rulebase and a single variable: *Find rules from the set of mined rules (Rulebase) whose LHS contains milk and cheese, whose support is greater than 5% and confidence is greater than 60%, and whose number of items on the LHS of the rule is fewer than 5.*

```
SELECT r
FROM Rulebase r
WHERE {milk, cheese} CONTAINED_EQ_IN LHS(r)
          AND SUP(r) > 5% AND CONF(r) > 60%
          AND SIZE_OF(LHS(r)) < 5
```

This query used function SIZE_OF, which is not in ***RC***.

Query 2. Minimal rules: *Find every rule from the set of mined rules (Rulebase) whose LHS and RHS do not contain the LHS and RHS of any other rule respectively.*

```
SELECT r
FROM   Rulebase r
WHERE NOT EXISTS
        (SELECT r′
         FROM Rulebase  r′
         WHERE r NOT_EQUAL r′ AND
                LHS(r′) CONTAINED_EQ_IN LHS(r) AND
                RHS(r′) CONTAINED_EQ_IN RHS(r)
```

We can also define maximal rules in Rule-QL in a similar way. Unlike a minimal rule that constitutes a most general rule in a rulebase, a maximal rule constitutes a most specific rule, i.e., its LHS and RHS are not contained in the LHS and the RHS of any other rule respectively.

Query 3. Unexpected rules (exceptions): *Given a set of mined rules (Rulebase) and a set of expected rules (ExpectedRules), find all unexpected rules in Rulebase with longer LHS and different RHS.*

```
SELECT DISTINCT r
FROM Rulebase r, ExpectedRules e
WHERE   LHS(e) CONTAINED_EQ_IN LHS(r) AND
        RHS(e) NOT_EQUAL RHS(e)
```

Query 4. Strong rules over time: *Find rules existing in all past 6 months* (M1-M6) *with confidence over 60%.*

```
SELECT r1
FROM M1 r1, M2 r2, M3 r3, M4 r4, M5 r5, M6 r6
WHERE   LHS(r1) EQUAL LHS(r2) AND
        RHS(r1) EQUAL RHS(r2) AND
        LHS(r1) EQUAL LHS(r3) AND
        RHS(r1) EQUAL RHS(r3) AND
        LHS(r1) EQUAL LHS(r4) AND
        RHS(r1) EQUAL RHS(r4) AND
        LHS(r1) EQUAL LHS(r5) AND
        RHS(r1) EQUAL RHS(r5) AND
        LHS(r1) EQUAL LHS(r6) AND
        RHS(r1) EQUAL RHS(r6) AND
        CONF(r1) > 60% AND CONF(r2) > 60%
        AND CONF(r3) > 60% AND
        CONF(r4) > 60% AND CONF(r5) > 60%
        AND CONF(r6) > 60%
```

Additional features of Rule-QL include support for statistical functions, the UNION operator, and support for GROUP BY, HAVING and ORDER clauses. We present these features now.

Statistical functions: Rule-QL supports various statistical functions on confidences and supports of rules, including:

- STABLE_RULES(r_1, ..., r_n): This function takes the rules r_1

through r_n and returns TRUE if the confidences and supports of these rules are "stable," where stable can be defined in various ways but with the same idea that they do not "significantly deviate" from each other. The language Rule-QL does not provide any particular definition of the concept of stability and leaves it to specific implementations to provide their own definition (or a collection of different definitions).

- INCREASING(a_1, ..., a_n) and DECREASING(a_1, ..., a_n), where a_1, ..., a_n are real numbers: These functions determine if values a_1, ..., a_n are increasing or decreasing respectively. Typically these values are confidences and supports of rules. In addition to these standard definitions, we can provide more sophisticated variations of these concepts (e.g., "most" of the values are increasing or decreasing).

Applicability of these functions is illustrated with the following examples that assume that we have 6 monthly rulebases, M1, M2, M3, M4, M5, and M6.

Query 5. Stable rules: *Find all the rules that have been stable over the 6 months.*

```
SELECT r1
FROM M1 r1, M2 r2, M3 r3, M4 r4, M5 r5, M6 r6
WHERE  LHS(r1) EQUAL LHS(r2) AND
       RHS(r1) EQUAL RHS(r2) AND
       LHS(r1) EQUAL LHS(r3) AND
       RHS(r1) EQUAL RHS(r3) AND
       LHS(r1) EQUAL LHS(r4) AND
       RHS(r1) EQUAL RHS(r4) AND
       LHS(r1) EQUAL LHS(r5) AND
       RHS(r1) EQUAL RHS(r5) AND
       LHS(r1) EQUAL LHS(r6) AND
       RHS(r1) EQUAL RHS(r6) AND
       STABLE_RULES(r1, r2, r3, r4, r5, r6)
```

Query 6. Trend rules: *Find all the rules whose confidences have been increasing over the 6 months.*

```
SELECT r1
FROM M1 r1, M2 r2, M3 r3, M4 r4, M5 r5, M6 r6
WHERE  LHS(r1) EQUAL LHS(r2) AND
       RHS(r1) EQUAL RHS(r2) AND
       LHS(r1) EQUAL LHS(r3) AND
       RHS(r1) EQUAL RHS(r3) AND
       LHS(r1) EQUAL LHS(r4) AND
       RHS(r1) EQUAL RHS(r4) AND
       LHS(r1) EQUAL LHS(r5) AND
       RHS(r1) EQUAL RHS(r5) AND
       LHS(r1) EQUAL LHS(r6) AND
       RHS(r1) EQUAL RHS(r6) AND
       INCREASING(CONF(r1), CONF(r2), CONF(r3),
          CONF(r4), CONF(r5), CONF(r6))
```

UNION Operator: When working with multiple rulebases with the same schema, it is often important to combine them together into one rulebase and define a variable over the combined rulebase. This functionality is supported in Rule-QL via the UNION operator.

UNION [ALL] (*RULEBASE*$_1$, ..., *RULEBASE*$_n$): This construct takes the rulebases *RULEBASE*$_1$ through *RULEBASE*$_n$ and forms their union, assuming that these rulebases are defined over the same itemset *I*. By default, no duplicate rules are returned as the output of the UNION operator. However, if the keyword ALL is added to the UNION keyword, then all the instances of duplicate rules from *RULEBASE*$_1$, ..., *RULEBASE*$_n$ are returned (as in the case of SQL [5]).

Query 7 below demonstrates how UNION is used in Rule-QL.

Grouping and Ordering Rules*:* As in SQL, we also group rules based on various grouping criteria and apply aggregate functions to individual groups. This capability would allow us to express such queries as

Query 7: *Find all the rules whose confidence is greater than 80% in at least 50% of the months.*

```
SELECT r
FROM r IN UNION ALL (M1, M2, M3, M4, M5, M6)
WHERE CONF(r) > 80%
GROUP BY RULE
HAVING COUNT > 3
```

Using the UNION operator, this query combines all the rulebases into one and then selects only the rules with confidence greater than 80%. Then, using the GROUP BY RULE operator, the query groups the resulting rules based on the *structure* of the rule (all the rules with the same structure, same LHS and same RHS, are put into one group). It retains only those groups having more than 3 rules in them (50% out of the total of 6 months).

Query 8: *Find average confidences of rules that exist in all the months.*

```
SELECT AVG_CONF(r)
FROM r IN UNION ALL (M1, M2, M3, M4, M5, M6)
GROUP BY RULE
HAVING COUNT = 6
```

In addition to GROUP BY RULE operator, Rule-QL supports GROUP BY ITEMSET, GROUP BY CONFIDENCE_RANGE and GROUP BY SUPPORT_RANGE operators that allow rule groupings based on a specified set of items and confidence and support ranges. Due to the space limitation, we do not describe the specifics of these operators in the paper.

HAVING clause is similar to the HAVING clause of SQL. It eliminates all the groups for which the conditional expression of the HAVING clause does not evaluate to TRUE. Finally, the ORDER clause can order rules based on their lengths (LHS/RHS of the rule, or both) and on their levels of confidence and support.

We complete this section with the following theorem, the proof of which is somewhat similar to the proof that SQL is more expressive than the relational calculus and is omitted because of the space limitation.

Theorem: Rule-QL is more expressive than calculus ***RC.***

5. QUERY EVALUATION

A naïve evaluation of Rule-QL queries has the following steps:

1. Compute the cross-product of the rulebases in the FROM clause. This produces a table of rules. Each row of the table is a list of rules and each column gives rules from one rulebase.
2. Delete those rows in the cross-product that fail the qualification conditions in the WHERE clause.
3. Project the table produced in Step 2 on the output rule variable specified in the SELECT clause.

4. If an aggregation function is specified in the SELECT clause, apply it to the result of Step 3.

This conceptual strategy is obviously very inefficient. We present a much more efficient query evaluation method in this section. However, by no means, we claim that the proposed techniques are the most efficient ones. We believe that subsequent work will produce more efficient Rule-QL query evaluation methods.

5.1. Indexing

Query evaluation algorithms need to retrieve rules from one or more rulebases. This can be achieved either by scanning (some) rulebases or accessing specific rules in the rulebases through indexes. We use the combination of both methods in Rule-QL. Below, we discuss the indexing schemes used in Rule-QL.

In Rule-QL, we use two kinds of indexing, i.e., inverted lists and B+ trees. Inverted lists are used to index rules using their items, while B+ trees are used to index the supports and confidences of the rules. Below, we discuss the inverted lists indexing. B+ trees will not be described, as they are well known.

Inverted lists: Inverted lists are a common indexing method used in text information retrieval [9,30], where each document is regarded as a set of keywords or items. This indexing was shown to be superior to many other indexing schemes for text retrieval [30]. They are very efficient for set-oriented operations and also fast to build. In association rules, LHS or RHS of a rule are sets of items. Query operations in Rule-QL are mainly set-oriented. Thus, inverted lists are a suitable indexing technique for rules.

Inverted lists indexing works as follows: Assume we have a set of rules $R = \{r_1, r_2, \ldots, r_m\}$, and each rule has a unique identifier (ID). We can index the LHS and/or the RHS of the rules according to the needs. Inverted lists index [9] has two parts: a vocabulary, containing all the distinct items in the set of rules (LHS or RHS); and for each distinct item an (inverted) list storing the IDs of the rules containing the item. Queries are evaluated by fetching the inverted lists of the query items, and then processing them for various needs. For example, to search for the supersets (i.e., CONTAINED_IN) of a constant set $\{a, b, c\}$ on the LHS of rules in a rulebase we find the inverted lists of items a, b, and c in the inverted file and traverse the lists to find the common rule IDs.

5.2. Evaluating Queries Involving a Single Rule Variable

We now start query evaluation with the simple case where only a single rule variable appears in the WHERE clause (note that this implies that the query involves only a single rulebase). Moreover, we consider only conjunctive queries (conditions are connected by conjuncts in WHERE) [3].

To illustrate our query evaluation strategy, consider the following single conjunctive query:

```
SELECT r
FROM Rulebase  r
WHERE    {a, b} CONTAINED_EQ_IN LHS(r) AND
         SUP(r) > 5% AND SIZE_OF(r) <= 5
```

This query can be evaluated in the following 3 ways:

1. We can scan the entire Rulebase, check the conditions on each rule, and add the rules to the results if the conditions are met.
2. We can utilize the inverted lists to access only those rules in Rulebase satisfying the condition
   $\{a, b\}$ CONTAINED_EQ_IN LHS(r).
   That is, we only need to retrieve those rules that contain a and b on the LHS first, and then check whether they satisfy the support, and SIZE_OF conditions.
3. We can use B+ trees as indexes on the supports of the rules in Rulebase and retrieve only the rules from Rulebase satisfying SUP(r) > 5%. Then we can scan the resulting rules and check the other two conditions.

Each query evaluation plan has its strengths and weaknesses depending on the size of the Rulebase, the corresponding indexes, and the nature of the expressions in the WHERE clause. Since methods (1) and (3) are straightforward, we will focus in the rest of the section on the second method that evaluates the CONTAINED_EQ_IN predicates. Note that CONTAINED_IN, CONTAINS, CONTAINS_EQ and EQUAL are similar to CONTAINED_EQ_IN, and can be handled in the same way.

The algorithm for evaluating expression S CONTAINED_EQ_IN L|RHS(r), where S is a constant itemset and L|RHS represents LHS or RHS of a rule, is presented in Figure 1. The algorithm first retrieves the inverted lists of all the elements in S, and uses the smallest set as the candidate answer set, AnswerSet (line 1 and 2). It then performs intersection simultaneously with the rest of the lists. Search in each list is done using binary search. Note that some other optimizations can also be used [6]. For simplicity, we do not include them in the algorithm.

```
Algorithm: findSuperSets(S, L|RHS, R)
        /* e.g., S CONTAINED_EQ_IN L|RHS(r) */
1   Retrieve the L|RHS inverted lists of R for all items in S;
2   Sort all the retrieved lists by size;
3   AnswerSet ← the smallest list (or set);
5   $st_i$ ← 1, where $i$ represents every retrieved list except the
        smallest one;
4   for each element $e \in$ AnswerSet do
6       for every other list $L_i$ in increasing order by size do
7           Perform a binary search for $e$ in $L_i$ between $st_i$ and $|L_i|$;
8           If $e$ is not found then Remove $e$ from AnswerSet  end
9           $st_i$ ← the value of current location of the binary search
10      end
11  end
```

Figure 1: Find all superset rules of a constant set S

A related expression is L|RHS(r) CONTAINED_EQ_IN S. This condition is more complex to process because we are unable to directly use the index on L|RHS of the rulebase, R. There are two options: (1) scan the rules in R and for each rule we check whether its L|RHS is contained in S, or (2) find all subsets of S on the L|RHS of the rules in R. Which option to take depends on the size of S because S has $2^{|S|}$ subsets. It is often the case that the size of S is small, and thus the number of subsets is not too large. If this is indeed the case, we can use the inverted lists indexing on R (of course, we need to generate all subsets of S). For each subset S_i of S, we evaluate the expression LHS(r) EQUAL S_i.

The algorithm for handling LHS(r) EQUAL S_i (called *findEqualSets*) is similar to *findSuperSets* except that *findEqualSets* needs to check equality after the if-statement of line 8 (Figure 1). That is, we need to add '**else if** $|S| \neq |L|RHS(e)|$ **then** Remove e from AnswerSet **end**'. As we will see in Section 6, that

[3] This is not a big restriction as single range queries containing disjuncts can be reduced to the union of conjunctive queries as in SQL [RG00].

the algorithms *findSuperSets* and *findEqualSets* are very efficient.

We now return to our original query to handle the conditions "SUP(r) >= 5% AND SIZE_OF(r) <= 5". This is easy: after each rule is found with *findSuperSets*, we can simply check the 2 conditions on the rule.

Aggregate operations: As explained in Section 4, Rule-QL supports aggregate operations, e.g., MIN, MAX, SUM, COUNT, GOURP BY.

The basic algorithm for aggregate operators consists of scanning the entire rulebase and maintaining some running information about the scanned rules. It is similar to that in relational databases. The GROUP BY operation can be implemented using sorting or hashing as in relational databases (see [22] for more details).

5.3. Queries Involving Multiple Variables

As was pointed out in Section 4, queries in Rule-QL can involve multiple rulebases and multiple variables defined over these rulebases. The WHERE clauses of such queries often contain many conditions/operations. In general, queries involving several operations have many evaluation options, and finding an efficient execution plan constitutes a significant challenge. In this section, we focus on the class of conjunctive queries involving multiple variables and multiple CONTAINED_IN, CONTAINED_EQ_IN, CONTAINS, CONTAINS_EQ and EQUAL conditions. We present an efficient evaluation plan for such queries.

We first define three types of conditions according to the number of rule variables that they contain:

Unary conditions: Such a condition contains only a single rule variable, e.g., SUP(r) > 5%, and S CONTAINED_IN L|RHS(r), where S is a constant set.

Binary conditions: Such a condition contains 2 rule variables, e.g., L|RHS(r1) CONTAINED_EQ_IN L|RHS(r2).

n-ary conditions: Such a condition contains 3 or more rule variables, e.g., statistical functions involving more than 2 variables, such as STABLE_RULES(r1, …, rn).

Given a set of conjunctive conditions, we first rewrite them by replacing conditions CONTAINS and CONTAINS_EQ with the appropriate equivalent conditions CONTAINED_IN and CONTAINED_EQ_IN in an obvious way. Then we build a directed graph, which we call a *query graph,* as follows. Each node of the graph is a rule variable together with its associated Rulebase. Each edge between two rule variables represents the set *B* of all binary conditions containing the two variables. All the unary conditions containing a single variable are associated with the node of the variable. Let r1 and r2 be two rule variables. The direction of the edge between r1 and r2 is defined as follows:

```
        If B = ∅ then there is no arc between r1 and r2
Case1   else if ∃ 'L|RHS(r1) EQUAL L|RHS(r2)' ∈ B then
                  there are two possible directions, which are
                  represented with r1 ↔ r2.
Case2       elseif ∃ 'L|RHS(r1)CONTAINED_IN L|RHS(r2)'∈ B
                  or ∃ 'L|RHS(r1) CONTAINED_EQ_IN
                       L|RHS(r2)' ∈ B then
                  the direction of the edge is from r1 → r2
Case3       else the edge direction is r2 ⇔ r1 (the conditions
                  involve only SUP, CONF, SIZE_OF, etc)
```

Note that n-ary conditions are not included above, as they form hyper-graphs. They typically require join operations for query evaluation (see Queries 5 and 6 in Section 4). These three cases are evaluated as follows:

Case1: We use indexing to compute L|RHS(r1) EQUAL L|RHS(r2). The specific algorithm *evaluate*↔() is described below. We check the other conditions in *B* after each pair of rules are identified.

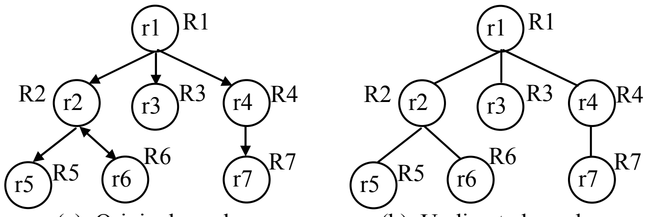
Case2: We use indexing to compute L|RHS(r1) CONTAINED_IN L|RHS(r2) or L|RHS(r1) CONTAINED_EQ_IN L|RHS(r2) and check the other conditions in B after each pair of rules from R1 and R2 are found. The algorithms that evaluate this condition, *evaluate*→ and *evaluate*←, are described below.

Case3: We use indexing on one condition, and check the rest. The algorithm, *evaluate*⇔, for evaluating this condition is also outlined below.

In query evaluation, we first evaluate the unary conditions in the WHERE clause and then proceed to the evaluation of the binary conditions described in Cases 1-3 above. Below, we present our evaluation method using the graph.

In general, query conditions involving two rule variables are analogous to join operations in relational databases. We also call them join operations. They are relatively expensive to evaluate. However, in many cases, we do not need full join of rulebases to evaluate Rule-QL queries. We show that an efficient evaluation method exists for a special class of queries whose query graph forms a *tree,* if the direction of each edge is removed, with the selection variable (from the SELECT clause) being its root. The part of the graph that is not connected to the selection variable can be ignored, as it does not affect the final answer. We believe that queries of this kind are the most common queries in practice.

Figure 2 shows an example. Figure 2(a) is the original graph, and Figure 2(b) is the undirected graph after the direction of each edge is removed. r1 is the selection variable. Note that each rule variable is also attached with its Rulebase (or its domain).



(a). Original graph (b). Undirected graph

Figure 2: An example conjunctive query graph

Figure 3 presents the overall algorithm for evaluating a tree query. Each node of the tree has a set of rules associated with the node that initially comes from the rulebase associated with the rule variable for the node. We traverse all the nodes of the tree in the post-order and recalculate the sets of rules for each node based on the conditions *B* associated with the edges between the nodes. More specifically, starting from each leaf node *N*, we work upward using the edge between the parent node *P* of *N* and *N*. We assume that we have already evaluated the set of rules associated with node *N* and we now compute the set of rules for node *P* using the conditions *B* for the edge *P-N.* After the edge *P-N* is evaluated, it is deleted. If *P* does not have any child node, it becomes a leaf node, and it is handled in the same way upward. The process goes on until it reaches the root node and all the branches of the root node are processed. The set of rules left in the root node is the answer set for the selection variable. Note that the algorithm traverses the tree only once.

```
Algorithm evaluateTree()
1  $Q \leftarrow$ every leaf node in tree;   /* e.g., the tree in Figure 2(b) */
2  while $Q$ not empty do
3     select a node $N$ from $Q$;   /* $N$ is not deleted from $Q$ yet */
4     if $N$ has no child node then
5        if $N$ has no parent node then   /* Reached the root */
6           $Q \leftarrow Q - \{N\}$;
7        else evaluateEdge($P$-$N$);  /* $P$ is the only parent of $N$ */
8           Remove edge $P$-$N$;
9           $Q \leftarrow Q \cup \{P\}$;
10 end
```

Figure 3: Evaluating a tree query

We use the tree in Figure 2(b) to illustrate how algorithm *evaluateTree* works. This tree has 3 leaf nodes initially, r5, r6 and r7, which are inserted into Q (line 1). We process r5 first (line 3). After evaluating the condition of the edge, r2-r5, we remove the edge from the tree. Node r2 is inserted into Q. However, we cannot handle this node because the edge r2-r6 has not been evaluated (line 4). After we finish r2-r6, r2 becomes available. This process goes on until we complete all the edges below r1.

evaluateEdge in *evaluateTree* evaluates each edge *P-N*. As stated above, there are three types of edges depending on the types of links between nodes P and N ($\leftrightarrow$, $\rightarrow$, $\Leftrightarrow$), and they have their own evaluation procedures which are described in turn below.

If the direction of the edge in the original graph is from its parent node P to N, i.e., $P \rightarrow N$, we use algorithm *evaluate*$\rightarrow$ in Figure 4. This is the most efficient edge evaluation algorithm, as we will see below. The algorithm is actually related to the procedure *findSuperSets* in Figure 1 except that: (1) procedure *evaluate*$\rightarrow$ has a loop to check every rule in P, while *findSuperSets* does not have the loop. S in *findSuperSets* is replaced with L|RHS1(P) in *evaluate*$\rightarrow$; and (2) for each rule r in P, we only need to find one superset rule in N, while *findSuperSets* finds all superset rules. This observation allows us to increase the efficiency of procedure *evaluate*$\rightarrow$ significantly.

```
Algorithm: evaluate→(L|RHS1, P, L|RHS2, N)
1  AnswerSet ← $P$;
2  for each rule $r$ in $P$ do
3     Retrieve the L|RHS2 inverted lists of rulebase N for all
         items of L|RHS1(r);
4     Sort all the retrieved lists by size;
5     FOUND ← false;
6     $st_i$ ← 1, where $i$ represents every retrieved list except
         the smallest one;
7     for each element $e$ in the smallest list do
8        for every other list $L_i$ in increasing order by size do
9           binary search for $e$ in $L_i$ between $st_i$ and $|L_i|$;
10          if $e$ is found then FOUND ← true
11          else remove $e$ from AnswerSet;
12               FOUND ← false;
13          $st_i$ ← the value of current location of the binary
                 search
14       end
15       if FOUND then exit for-loop end
16    end
17 end
```

Figure 4: Find rules in P whose L|RHS has an L|RHS superset rule in N

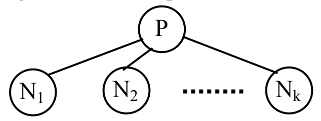
If the direction of the edge in the original graph is from a child node N to the parent node P, i.e., $P \leftarrow N$, we need to find all the superset rules in P for each rule in N. To do that, we can directly utilize the procedure *findSuperSets* in Figure 1 as the core part of the algorithm. More specifically, our procedure *evaluate*$\leftarrow$() scans the rules in N and for each rule r it utilizes the procedure *findSuperSets*(L|RHS(r), L|RHS, P) to determine such supersets. We do not present algorithm *evaluate*$\leftarrow$ as it is a straightforward extension of the algorithm *findSuperSets* from Figure 1. One simple optimization to *findSuperSets*, however, is to skip a rule in P (or R in Figure 1) if it is already in the answer set.

If the direction of the edge in the original graph is $P \leftrightarrow N$, the corresponding evaluation algorithm *evaluate*$\leftrightarrow$ is very similar to algorithm *evaluate*$\rightarrow$ presented in Figure 4 with one modification: after each e is found in line 10, we need to do an additional check for size equality. If the direction of the edge in the original graph is $P \Leftrightarrow N$, the procedure *evaluate* $\Leftrightarrow$ can handle it similarly.

It is easy to see that *findSuperSets*, *findEqualSets* and *evaluate*$\rightarrow$ are correct. We now state that our overall algorithm for evaluating a tree query in Figure 3 is correct.

Theorem: The evaluateTree algorithm is correct.

Proof: We only need to show that a tree with only two levels can produce the required result at the parent node. The figure below gives such a tree, a parent node P with k leaf nodes, $N_1, N_2, \ldots, N_k$. When a tree with multiple levels, the situation is the same, as the algorithm in Figure 3 works upward and deletes edges along the way.



For the 2-level tree above, we process the edges one by one. Let us start with edge P-N_1. After it is evaluated, we know that all the remaining rules in P have corresponding rules in N_1 that satisfy the conditions of edge P-N_1. We then process P-N_2, which reduces the set of rules in P further. The remaining rules in P must now also satisfy the conditions of edge P-N_2. Since we only reduce the size of P but not N_1, the remaining rules in P still meet the conditions of P-N_1. This goes on until we finish P-N_k. Clearly, after P-N_k, all the remaining rules in P satisfy the conditions associated with P-N_1, …, and P-N_k. It is important to note that not all rules in N_1, N_2, …, N_{k-1} may satisfy the edge conditions after all the edges are evaluated, but that does not matter because none of them is our selection-variable. Only P is a possible selection variable. After the set of rules in P are decided, we can work upward in the same fashion if P is not the root to obtain the correct answer at the root.

We now discuss why the overall algorithm in Figure 3 is efficient. If we do not start from edges connected to leaf nodes and work upward, but from other edges, we may need to evaluate each edge multiple times. For example, in Figure 2(b), if we first work on the edge r1-r2 and then on edge r2-r4, we reduce the rules in r2 and r4. However, some of the rules in r1 may not be valid any more because the effect of r2-r4 evaluation needs to be propagated to all related edges. Thus, we need to evaluate r1-r2 again. In general, the propagation effect can be very serious, which results in the same edge being evaluated multiple times. In contrast, if we traverse the tree in post-order, we do not need to re-visit any edges as after being processed, the low-level nodes do not affect the final results at the root.

We now turn to the general case when the query graph contains cycles. In such cases, we can handle those sub-graphs that are trees in the same way, and those edges in the cycles using traditional join operations. Figure 5 shows an example graph (r1 is the selection-variable). We can first use the algorithm *evaluateTree* to evaluate the sub-graph involving r2, r5 and r6, and also the sub-graph involving r1, r4 and r7. We can then handle the cycle r1-r2-r3-r1 using traditional join techniques.

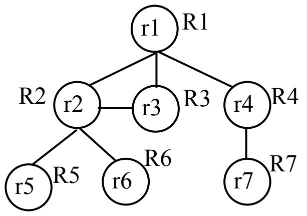


Figure 5: An example graph with a cycle

As a final note, we state that all the rulebases and inverted lists can be stored on disk. This is similar to that in text document retrieval and Web search as a text document is represented as a bag of words (or items). Techniques from that research can be borrowed for use in our case. See [30] for more details on how to store and retrieve inverted lists into memory from disk.

6. EXPERIMENT RESULTS

This section reports our experiment results to show the feasibility of Rule-QL. We have carried out a prototype implementation of the fragment of Rule-QL as described in Section 5. We tested the performance of our system on the first five representative queries presented in Section 4.

Note that we write Query 4 in two ways. The first method forms a query tree with two levels (Figure 6(a)). The leaf level has all the rule variables as the children of the root r1. This query is shown in Section 4 (denoted as Q4a). The second query forms a long chain graph (also a tree but with 6 levels) starting from r1 (Figure 6(b)). This query is given below, denoted as Q4b.

```
SELECT r1
FROM M1 r1, M2 r2, M3 r3, M4 r4, M5 r5, M6 r6
WHERE   LHS(r1) EQUAL LHS(r2) AND
        RHS(r1) EQUAL RHS(r2) AND
        LHS(r2) EQUAL LHS(r3) AND
        RHS(r2) EQUAL RHS(r3) AND
        LHS(r3) EQUAL LHS(r4) AND
        RHS(r3) EQUAL RHS(r4) AND
        LHS(r4) EQUAL LHS(r5) AND
        RHS(r4) EQUAL RHS(r5) AND
        LHS(r5) EQUAL LHS(r5) AND
        RHS(r5) EQUAL RHS(r5) AND
        CONF(r1)>60% AND CONF(r2)>60% AND
        CONF(r3)>60% AND CONF(r4)>60% AND
        CONF(r5) > 60% AND CONF(r6) > 60%
```

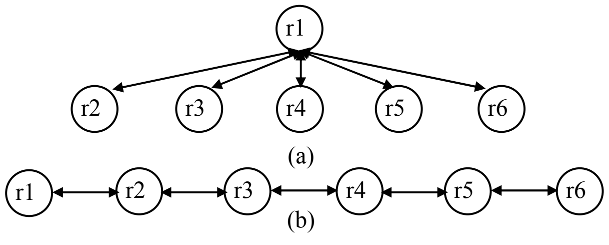


Figure 6: Two query trees of Query 4

The purpose of writing this query in two ways is to show the efficiency of the system with different types of tree queries. This query serves as a stress test of Rule-QL as it involves 6 rulebases and a large number of itemset comparison operations.

Note that the STABLE_RULES predicate in Query 5 is implemented using the Chi-square test [7] in our system.

The execution times of these queries are presented in Figure 7. Each execution time is the average result of 10 runs of the query, and includes the time for reading in rulebase(s) and the time for building appropriate indices (mainly inverted lists). Also, we kept the indices in the main memory. This is reasonable with modern computers, as the number of mined rules is usually moderately large compared with the number of tuples in a relational database. Our experiments were carried on a Pentium III 800 PC with 500MB of memory. For each query, we vary the number of rules in each rulebase from 10,000 to 100,000 (10k-100k). All the rules are mined from a dataset generated using the IBM data generator [2], which is commonly used in evaluating association rule mining algorithms. For our experiments, we first generate a large number of rules using a low minimum support. For each individual experiment we randomly select some rules to obtain the required number of rules for the experiment.

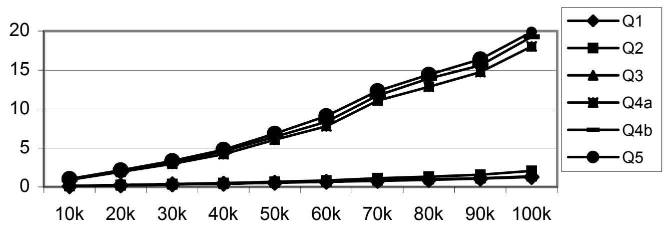


Figure 7: Execution times (in sec.) of the 5 representative queries with different number of rules

We next discuss how each query is evaluated and the results of our experiments.

Query 1. This query is evaluated using the algorithm in Figure 1 with additional checks on support, confidence and rule LHS size. From Figure 7, we observe that this query can be executed extremely efficiently.

Query 2. This query involves heavier computation than the first one due to the nested query. Each nested query finds one subset of the LHS of r on the LHS of a rule r′. The query is processed by looping on each rule in Rulebase and computing L|RHS(r) CONTAINED_EQ_IN S using the technique from Section 5.2. The index on Rulebase is built only once for all the nested queries. From Figure 7, we can see that the execution is extremely fast even with a large number of rules and relatively straightforward implementation of nested queries.

Query 3. In this set of experiments, we fix the number of expected rules to 20, as the user usually could not provide many expected rules. This query is processed directly using the algorithm in Figure 4. From Figure 7, it can be seen that the executions are very efficient, and scale up roughly linearly with the number of rules in Rulebase. It is easy to see that as the number of expected rules increases, the execution time also grows linearly (not shown in Figure 7).

Query 4. In this experiment, each rulebase has the same set of rules, and their confidences are all more than 60%. This means that there is no reduction in the number of rules in each operation. Thus, it represents the worst-case scenario. This query is evaluated using the algorithms *findEqualSets* and *evaluateTree* with additional checks on confidences. Figure 7 shows that the execution times are reasonable even with 100,000 rules in each rulebase. The execution times scale up linearly. At any one time, only one rulebase is in memory.

Query 5. As Query 4, this query also serves as a stress test for Rule-QL as it requires an expensive join operation with 6 rulebases. Join operation is required because we need the values of CONF(r1), CONF(r2), CONF(r3), CONF(r4), CONF(r5), and CONF(r6) for the stability test. As in Query 4, the 6 rulebases are the same. Thus, no reduction in the number of rules occurs in each operation. Like that in Query 4, this also represents the worst-case situation. This query is evaluated with join operations. Each join involves one rulebase (Rulebase_i) and the set of join results of the previous join operations. The join operation is performed as follows: We first read in Rulebase_i and build the inverted lists index. We then perform join based on the algorithm *findEqualSets* for each rule in the previous join results. In the join results, only one copy of the qualified rules is stored since we are doing an equal join. However, the support and confidence values of each rule from every rulebase are remembered. After each join operation, the results are written on disk. Figure 7 shows that the execution times are reasonable and scale up linearly. Again, at any one time, only a single rulebase is in memory.

In summary, our experimental results clearly showed that these queries could be processed very efficiently, which indicates the practical feasibility of Rule-QL.

7. CONCLUSIONS

Although several data mining algorithms often generate a large number of rules, the problem of post-analysis of the discovered rules has not been explored fully despite its importance. This paper addresses the problem by proposing a rule query language Rule-QL to facilitate ad hoc querying of the mined rules. We believe that the language will make the rules more accessible to the user and data mining results more useful in practice.

From a theoretical point of view, the paper presented a rule calculus (***RC***), which is based on first order logic defined over multiple rulebases. We proved that all the safe queries in ***RC*** can be expressed with Rule-QL queries. This makes Rule-QL more powerful than previously proposed rule query languages and allows it to express several practically important queries that could not be expressed before, including minimal and unexpected rule queries. We also discussed query processing issues for Rule-QL and presented several efficient query evaluation techniques. Experimental results on a diverse set of queries showed that Rule-QL is practically feasible. In the future work, we plan to develop more efficient query processing methods and also to integrate Rule-QL with a rule mining algorithm.

REFERENCES

[1]. Abiteboul, S., Hull, R. and Vianu, V. *Foundations of Databases,* Addison-Wesley, 1995.

[2]. Agrawal, R. and Srikant, R. "Fast algorithms for mining association rules." *VLDB-1994*.

[3]. Agrawal, R, Psaila, G., Wimmers, E., and Zait, M. "Querying shape of histories." *VLDB-95*.

[4]. Bayardo, R. and Agrawal, R. "Mining the most interesting rules" *KDD-99*, 1999.

[5]. Date C.J. and Darwen, H. *A Guide to the SQL Standard*. Third Edition, 1993.

[6]. Demaine, E., Lopez-Ortiz, A., and Munro, J. "Experiments on adaptive set intersections for text retrieval systems." *Proc. 3 Wrksh. on Alg. Engineering & Experiments*, 2001.

[7]. Everitt, B. S. *The analysis of contingency tables*. Chapman and Hall, 1977.

[8]. Faloutsos, C. "Signature files." In W. Frakes and R. Baeza-Yates, editors: *Information retrieval: data structures and algorithms*. Chapter 4, pp: 44-65, Prentice Hall, 1992.

[9]. Fox, E., Harman, D., Baeza-Yates, R., and Lee, W. "Inverted files." In W. Frakes and R. Baeza-Yates, editors: *Information retrieval: data structures and algorithms*. Chapter 3, pp: 28-43, Prentice Hall, 1992.

[10]. Han, J., Fu, Y., Wang, W., Koperski, K. and Zaiane, O. "DMQL: a data mining query language for relational databases." *SIGMOD Workshop on DMKD*, 1996.

[11]. Han, J., Lakshmanan, L. and Ng, R. "Constraint-based, multidimensional data mining" *IEEE Computer*, Aug. 1999.

[12]. Hilderman, R.J. and Hamilton, H.J. "Evaluation of Interestingness measures for ranking discovered Knowledge," *PAKDD-2001*, 2001.

[13]. Imielinski, T., and Mannila, H. "A database perspective on knowledge discovery." *CACM*, 39(11), 58-64, 1996.

[14]. Imielinski, T., Virmani A, Abdulghani, A. "DMajor-Appliction programming interface for database mining" *Journal of DMKD*, 1999.

[15]. Klemetinen, M., Mannila, H., Ronkainen, P., Toivonen, H., and Verkamo, A.I. "Finding interesting rules from large sets of discovered association rules." *CIKM-1994*, 1994.

[16]. Liu, B., Hsu, W., and Chen, S. "Using general impressions to analyze discovered classification rules." *KDD-97*, 1997.

[17]. Liu, B., Ma. Y & Lee. R. "Analyzing the interestingness of association rules from the temporal dimension" *ICDM-01*.

[18]. Meo, R. Psaila, G., and Ceri, S. "A new SQL-like operator for mining association rules," *VLDB-96*, 1996.

[19]. Morzy, T. and Zakrzewicz, M. "Group bitmap index: A structure for association rules retrieval." *KDD-98*.

[20]. Padmanabhan, B. and Tuzhilin, A. "A belief-driven method for discovering unexpected patterns." *KDD-98*, 1998.

[21]. Piatesky-Shapiro, G., and Matheus, C. "The interestingness of deviations." *KDD-94*, 1994.

[22]. Ramakrishnan, R. Gehrke, J. *Database Management Systems*. McGraw-Hill, 2000.

[23]. Shen, W-M, Ong, K-L, Mitbander, B., and Zaniolo, C. "Metagueries for data mining." In *Adv. in Knowledge Disc. and Data Mining* (eds: U.M. Fayyad, G. Piatetsky, P. Smyth, and R. Uthurusamy), AAAI/MIT Press, 1996.

[24]. Silberschatz, A., and Tuzhilin, A. "What makes patterns interesting in knowledge discovery systems." *IEEE Trans. on Know. and Data Eng.* 8(6), 1996, 970-974.

[25]. Suzuki, E., "Autonomous Discovery of Reliable Exception Rules." *KDD-97*, 1997.

[26]. Tan, P-N. & Kumar, V. "Interestingness measures for association patterns: a perspective." *KDD-2000 Workshop on Post-processing in ML and DM*, 2000.

[27]. Van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[28]. Virmani A., Imielinski, T. M-SQL: A query language for database mining." *Journal of DMKD*, 1999.

[29]. Zhong, N., Yao, Y.Y., Ohshima, M., and Ohsuga, S. "Interestingness, Peculiarity, and Multi-Database Mining", *IEEE ICDM*-01, 2001.

[30] Zobel, J., Moffat, J., & Ramamohanarao, K. "Inverted files versa signature files for text indexing." *ACM TODS,* 1998.