
Data Mining and Knowledge Discovery 3, 373–408 (1999)

MSQL: A Query Language for Database Mining

TOMASZ IMIELIŃSKI imielins@cs.rutgers.edu
AASHU VIRMANI avirmani@cs.rutgers.edu
Rutgers University, Department of Computer Science,
New Brunswick, NJ 08903 USA

Editors: Fayyad, Mannila, Ramakrishnan

Abstract. The tremendous number of rules generated in the mining process makes it necessary for any good data mining system to provide for powerful query primitives to post-process the generated rulebase, as well as for performing selective, query based generation. In this paper, we present the design and compilation of MSQL, the rule query language developed as part of the *Discovery Board* system.

Keywords: database mining, query language, MSQL, SQL, association rules

1. Introduction

In Imielinski and Mannila (1996) we have argued that due to the tremendous number of association rules which are generated by data mining there is a pressing need to provide rule querying, both to selectively generate rules from data as well as query the rulebase of rules which were previously generated. Several such query languages have been proposed in the literature (Han et al., 1996; Meo et al., 1996). Other similar efforts were also recently noticed in literature, either to generate rules satisfying some constraints in the database (Ng et al., 1998) or to integrate mining closely with relational databases (Sarawagi et al., 1998).

In our view, although these proposals are a step in the right direction, they still lack a few features such as closure, using the full expressive power of SQL, and treating rule querying in a manner similar to rule generation. Following is the set of primitives which, in our opinion, a rule query language should satisfy:

- *Ability to nest SQL:* Since SQL is a well accepted interface to relational databases, and allows for declarative, set level, expressions which are optimized substantially, it is desirable if users can utilize SQL primitives such as sorting, group-by, and others like these within MSQL, and be able to express nested queries using the SQL nested query constructs like [NOT] IN, [NOT] EXISTS etc.
- *Closure:* Mining is essentially an iterative task, which often involves refinement of existing queries and regenerating results. The language must provide for operations to further manipulate the results of the previous MSQL query.
- *Cross-over between data and rules:* To permit the "iterative refinement" as mentioned in the previous item, the language must allow primitives which can map generated rules back to the source data, and vice-versa.

- *Generation versus querying:* Given the enormous size of rulebases, it may not always be possible to "extensionally" maintain rulebases. The language should allow a user to express rule-generation vs. rule-querying using the same expression syntax.

Keeping the above guidelines in mind, MSQL, the language developed in (Virmani, 1998) starts with the SQL92 standard and adds support for rule-manipulation operations in a familiar SQL-like syntax. Due to space restrictions, we present an overview and some examples of the query language here. For a more complete treatment of the language, including its evaluation and optimizations, we refer the reader to the above reference.

2. Basic notions

Query based rule generation, and successive refinement of the original mining task through the use of a series of rule queries form an important part of knowledge discovery using rules. The two main commands in MSQL thus deal with the activities of **data-mining** and **rule-mining**. We define data-mining as the process of generating rules from the data in response to a query, and rule-mining as the process of querying a pre-existing rulebase. The **GetRules** and **SelectRules** commands in MSQL perform these two activities respectively. Below we introduce some terminology and basic notions that are used throughout the rest of the paper.

Definition 1. A **descriptor** is an expression of the form $(A_i = a_{ij})$, where a_{ij} belongs to the domain of A_i. For continuous valued attributes, a descriptor of the form $(A_i = [lo, hi])$ is allowed, where $[lo, hi]$ represents a range of values over the domain of A_i.

Definition 2. A **conjunctset** stands for a conjunction of an arbitrary number of descriptors, such that no two descriptors are formed using the same attribute. The **length** of a conjunctset is the number of descriptors which form the conjunctset. A descriptor is thus the special case of a singleton conjunctset.

Definition 3. A record (tuple) in R is said to **satisfy** a descriptor $(A_i, = v_{ij})$, if the value of A_i in the record equals v_{ij}. To satisfy a conjunctset, a record must satisfy **all** k descriptors forming the conjunction.

Example: Let R be a relation represented by the table shown below:

EmpId	Job	Sex	Car
1	Doctor	Male	BMW
2	Lawyer	Female	Lexus
3	Consultant	Male	Toyota
4	Doctor	Male	Volvo

Then (**Job** = `Doctor`) is an example of a descriptor in the above data, satisfied by records with EmpId values 1 and 4. Along the same lines, (**Sex** = `Female`) $\wedge$ (**Car** = `Lexus`) is an example of a conjunctset of length 2 in the above data.

Definition 4. By a **propositional rule** over R, we mean a tuple of the form $\langle B, C, s, c\rangle$, where B is a conjunctset called the **Body** of the rule, C is a descriptor called the **Consequent** of the rule, s is an integer called the **support** of the rule, and c is a number between 0 and 1 called the **confidence** of the rule. Support is defined as the number of tuples in R which satisfy the body of the rule, and **confidence** is defined as the ratio of the number of tuples satisfying both the body and the consequent to the number of tuples which satisfy just the body of the rule.

Intuitively, rules are if-then statements of the form "if Body then Consequent", with s and c being their quality measures computed in R. We will usually represent rules in the following syntactic form:

$$Body \Rightarrow Consequent \quad [support, confidence]$$

For instance, the following is a rule over the example relation shown earlier:

$$(\textbf{Job} = Doctor) \wedge (\textbf{Sex} = Male) \Longrightarrow (\textbf{Car} = BMW) \quad [2, 0.5]$$

A rule in our case is thus a generalization of the association discussed in (Agrawal et al., 1993). Since we allow user defined procedures and functions in our API, the expressive power of propositional rules is actually, for all practical reasons equivalent to non-recursive predicate rules.

A rule can also be viewed as a query when applied to a relation. We say a relation R **satisfies** a rule $r = \langle B, C, s, c\rangle$ if there are at least s tuples in R which satisfy B and at least a fraction c of them satisfy the conjunction $B \wedge C$. This is also expressed by saying that r **holds** in R. If R does not satisfy r, then we say that R **violates** r, or alternately, r **does not hold** in R.

Since rules represent aggregates over a set of tuples, the relationship between a rule and an individual tuple cannot be similarly defined. However if we only consider the rule-pattern $\langle B, C\rangle$ without the associated support and confidence, we can define the following relationships between them.

A tuple t **satisfies** a rule pattern $\langle B, C\rangle$, if it satisfies the conjunction $B \wedge C$, and it **violates** the above pattern if it satisfies B, but not C.

3. Language syntax

The MSQL syntax is comprised of four basic statements. The main intuition behind the language design has been to allow representation and manipulation of rule components, mainly Body and Consequent, which, being sets, are not easily representable in standard SQL.

The outline of the syntax for these MSQL extension is shown below.

```
<MSQL Stmt> ::=         <GetRules-Query>
              |         <SelectRules-Query>
              |         <Sat-Violate-SubQuery>
              |         <Encode-Stmt>
```

We consider each MSQL command in its respective section. For complete expansions of any individual statement syntax, refer to Virmani (1998). A quick overview of these constructs is as follows. The Encode statement provides pre- and post-processing support and is discussed in Section 6. The GetRules query is used for rule-generation, and the SelectRules query, which follows the same syntax (covered in GetRules-Query syntax), is used to query rules from an existing rulebase. In addition, a standard SQL query on a database table can have a nested GetRules sub-query in its "where" clause connected via the Satisfy or Violate keyword. Syntax for this clause is covered under the Sat-Violate-SubQuery statement.

4. General query syntax

The most general formulation of the GetRules Query is as follows:

```
[Project Body, Consequent, confidence, support]
GetRules(C) [as R1]
[into <rulebase_name>]
[where <conds>]
[sql-group-by clause]
[using-clause]
```

where C is a database table, and R1 is an alias for the rulebase thus generated. In addition, ⟨Conds⟩ may itself contain:

```
<Rule Format Conditions RC> |
<Pruning Conditions PC> |
<Mutex Conditions MC> |
<Stratified Subquery Conditions SSQ> |
<Correlated Subquery Conditions CSQ>
```

The GetRules operator generates rules over elements of the argument class C, satisfying the conditions described in the "where" clause. The results are placed into a rule class optionally named by the user, else named by suffixing 'RB' to the name of the source class. (So for instance, the class Emp generates the rulebase EmpRB). The projection and group-by operations can optionally be applied, and their meaning is the same as defined in SQL. Since they basically post-process the generated rules, they do not affect the semantics of rule generation.

The GetRules operator, as considered here, generates rules among discrete attributes only. Continuous attributes are supposed to be have been discretized off-line, using user defined attributes, or the encode command described later.

Another important thing to note is that the GetRules query operates on the complete class C, rather than a subset of it. There is a difference. All rules from the subset of data with $(A_1 = a)$ in them is not the same as all rules on the whole data with $(A_1 = a)$ in the Body, since if we subset and then mine for rules, the confidence and support in the rules generated will change. Besides, if one mines for rules about a subset of the data, then technically, it is a different class and therefore, there should be a different rulebase corresponding to it.

Given the above reasoning, the GetRules operator disallows any "where" clause conditions on pure attributes of the source class C. These can always be performed by creating a view on C with the appropriate selections/projections and then using GetRules on the view. The only conditions allowed are the ones on rule components: Body and Consequent. Note that the evaluation of GetRules internally may involve selecting/projecting the data for efficiency, but it will preserve the query semantics.

4.1. Conditions in the "where" clause

To understand the complete syntax of the query, let us first examine the different types of conditions possible in the "where" clause. They have been categorized in the following groups to facilitate understanding of the evaluation plan. Each type of condition affects the execution plan in a different way.

- ⟨**Rule Format Conditions RC**⟩ Rule format conditions occur on the Body and the Consequent, and have the following format:

```
Body { in | has | is } <descriptor-list>
Consequent { in | is } <descriptor-list>
```

  The operators **in, has** and **is** represent the subset, superset and set-equality conditions respectively. A few examples below explain their usage. Note that one or both operands could be constants.

```
r.body has {(Job='Doctor'), (State='NJ')}
r1.consequent in r2.body
r.body in {(Age=[30,40]), (Sex='Male'), (State=*)}
r.consequent is {(Age=*)}
```

  The first condition above is true for all rules which have at least the predicate `(Job='Doctor')` $\wedge$ `(State='NJ')` in their bodies. Similarly, the second condition above is true for all pairs of rules (r1, r2), such that r1's Consequent is an element of r2's Body. The third clause above, when present in a GetRules query, specifies that the bodies of rules produced may only belong to a subset of the set of combinations generated by the expression "`(Age = [30,40]) (Sex='Male') (State =*)`". The fourth condition

restricts the Consequent attribute to Age, without restricting its values. In general, the "where" expression can have several of these conditions connected via AND and OR logical operators.

Definition 1. The expression $(A_i = *)$, where A_i is any method in C, is a descriptor-pattern representing a family of descriptors of the type $(A_i = a_{ij})$.

In MSQL, a descriptor pattern can be used anywhere a descriptor is allowed. In the above case of $(Age = *)$, any descriptor of the form $(Age = [lo, hi])$ satisfies the pattern.

- ⟨**Pruning Conditions PC**⟩ These are conditions involving support, confidence and lengths of Body and Consequent, which can be used to control the algorithm. These have the format:

```
confidence <relop> <float-val in [0.0,1.0]>
support <relop> <integer>
support <relop> <float-val in [0.0,1.0]>
length <relop> <integer>
relop ::= { < | <= | = | >= | > }
```

Confidence is specified as a fraction between 0 and 1, while support can be specified either as a fraction of the database, or as an absolute number of records.

- ⟨**Mutex Conditions MC**⟩ These conditions define sets of two or more attributes such that, in a given rule, if there is an attribute from one of these sets, then that rule doesn't have any other attribute from that set. The syntax is:

```
Where <other-conditions>
      { AND | OR } mutex(method, method [, method])
     [{ AND | OR } mutex(method, method [, method])]
```

For instance, the condition "`mutex(zipcode, county, phone_area_code)`" can be used in the "where" clause of the GetRules operator to avoid expansion of any rule containing one of the above attributes with another attribute from the same set.

- ⟨**Subquery Conditions SSQ**⟩ These are subqueries which are connected to the "where" clause using either EXISTS, NOT EXISTS, or the IN connectives. The connectives used to join outer and inner queries preserve their SQL semantics. Subqueries are dealt with in more detail in Section 5.2.

5. Generating and retrieving rules

All examples in this section assume the presence of data about employees in the following schema:

```
Emp(emp_id, age, sex, salary, nationality, position, smoker, car)
```

To generate rules from the Employee table, one uses the GetRules command as follows:

```
GetRules(Emp)
where Consequent in { (Age=*), (Salary=[30000,80000]) }
and Body has { (Job=*) }
and confidence > 0.3
and support > 0.1
```

The above query would generate rules having at least Job in the antecedent and either Age or Salary as the Consequent methods with values of salary within the ranges specified. Also note that there is an implicit line:

```
Project Body, Confidence, Confidence(Emp), support(Emp)
```

before the GetRules command. Later we will show how this can be altered to evaluate existing rules on different sets.

The typical mechanism observed to be most commonly used in query based mining is for users to first do a fairly general GetRules query and store the result persistently, and to follow this with a series of SelectRules queries, each of which selects a small subset of the entire rulebase.

To generate rules for a given database table, the **GetRules** operator must be used with a table argument as follows:

```
GetRules(T)
into R
where confidence > 0.3
and support > 0.1
```

This will generate all rules existing in table T matching the confidence and support requirements, and put them in a persistent rulebase named *R*.

For future selections of these rules, the language has the **SelectRules** command. SelectRules will not generate any new rules, but rather rely on the contents of the argument rulebase for providing results. For instance, the following query retrieves rules with at least Age and Sex in the Body and the car driven as a Consequent.

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

Note that by default, the lowest confidence and support of the rules produced by this query will be 30 percent and 10 percent respectively, since those were the parameters R was mined with.

So far we have used the Project operator implicitly. One can use the Projection to explicitly evaluate different rule patterns over various similar databases. For instance, if NJ_Emp and NY_Emp are two views defined on the Emp table, one might be interested in knowing how the two data sets compare with respect to the above rule pattern. The following query, in this case,

```
Project Body, Consequent, Confidence(NJ_Emp), Support(NJ_Emp),
                          Confidence(NY_Emp), Support(NY_Emp),
SelectRules(R)
where Body has { (Age=*), (Sex=*) }
  and Consequent is { (Car=*) }
```

will select the Bodies and Consequents of rules from R, and evaluate them over the two views. SelectRules, by definition, *does not* generate new rule patterns.

The above example brings up an interesting issue: What if R is not a rulebase generated by the Emp table, but rather, by some other table? There are two possible scenarios. In the simpler case, R could be a rulebase not containing the attributes required in the query (in this case, Age, Sex and Car). In that case, the query will be syntactically incorrect and will return an error. In a more complicated case, the rule table and the other data tables in the above type of SelectRules query could both contain the attributes required by the "where" clause, even when they semantically meant something totally different. Should the language be enforcing this "typing" between rulebases and databases?

Our design decision has been to follow strong typing between rulebases and databases in our API, implemented in C++, but to treat both rulebases and datasets as untyped relational tables in MSQL. In a typical relational environment, it is quite possible for someone to join two tables A and B on two integer fields A.length and B.speed thus yielding a senseless result. Similarly, it is extremely hard to police the proper use of these operators once the queries span more than one table. The onus in both cases lies on the user and the data dictionary to ensure correct semantics in the operations. In Virmani (1998), we describe our proposed catalog enhancements to ensure that there is enough metadata kept in the DBMS catalogs to allow the user to correctly identify the rulebases developed from a given data table.

5.1. Using satisfy and violate

An extremely important feature in a database mining system is the ability to correlate the generated knowledge (rules in this case) with the source data which produced that knowledge. The **Satisfy** and **Violate** operators in MSQL provide this capability of connecting the rules with the data.

Both Satisfy and Violate operators take a tuple and a set of rules and return true if the tuple satisfies (violates) any (or all) the rules in the set. Syntactically, their usage is much like the EXISTS operator in SQL:

```
Select ...
From   ...
Where { SATISFIES | VIOLATES } { ALL | ANY } (
                GetRules | SelectRules Subquery
        )
```

Example 1. Find all records in Emp table which violate all rules of the form "`age` $\Rightarrow$ `salary`" above a 30% confidence.

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

5.2. Nesting the GetRules operator

The Satisfy and Violate connectives discussed above show one form of nesting within GetRules queries. Interesting queries can also be formulated using SQL-like nesting of multiple GetRules queries. The semantics for such queries is exactly the same as in SQL if we first generate both the rulebases in the inner and outer query, and treat them as two classes in SQL. (However, this may not be the best approach for evaluation, as will be shown later.)

For evaluation purposes, one can distinguish nested queries in which the inner query makes a reference to the data or rule objects of the outer query, from queries where there is no cross-referencing of data. Using SQL terminology, the former can be called **correlated queries**, and the latter, **stratified queries**. For instance, the following is an example of a correlated query, since the rule class generated in the outer query, R1, is referenced in the inner query.

```
GetRules(C) R1
where <pruning-conds>
and not exists ( GetRules(C) R2
                 where <same pruning-conds>
                 and R2.Body HAS R1.Body )
```

The above query finds rules that are "on the border", i.e. if they are expanded any more, they will drop out of support. The next query is an example of nesting a SelectRules within a GetRules operator. It generates all rules where the Consequent exists in a prior set of rules. It is stratified, since no references to R1 are made in the nested subquery.

```
GetRules(C) R1
where <pruning-conds>
  and Consequent is {(X=*)}
  and Consequent in ( SelectRules(R2)
                      where Consequent is {(X=*)} )
```

6. Pre/Post-processing enhancements

Although GetRules and SelectRules form the basic core around which MSQL was designed, there is one other extension to the language that we felt was basic enough to be included

as part of the core. Although the operation presented in this section can be performed by mapping the data using an external table, the processing required by the end user will be complex, and the results will not be as smoothly integrateable with the other language commands.

6.1. Preprocessing continuous valued attributes

Continuous valued attributes in the original data must first be discretized or "binned" into intervals before mining can performed on them, to ensure that they will reach a sufficient level of support. (It is less likely for an individual value from a continuous range to gather enough support in real data sets.) A fairly straightforward way to preprocess such attributes is to create a new table, with all continuous values replaced with a discrete identifier which stands for an interval and then apply GetRules to this modified table. However, there are a few disadvantages with this approach:

- It is time and space consuming to create a separate copy of a large table.
- Different users may want different discretizations for the same attribute.
- When using Satisfy/Violate primitives, the resulting tables will have interval identifiers, which make the results harder to interpret, not to mention the information loss due to any kind of discretization.

The Encode operator solves most of the above problems by effectively telling the mining program to do an "on the fly" discretization. No separate copy of the table is needed, and different users can share, or specify their own discretization functions on various attributes. The results of satisfy and violate present the original table with no modifications, which is better for future mining.

The syntax for the create encoding command effectively creates ranges of values, and assigns discrete integers to those ranges. MIN and MAX stand for the lowest and highest values of the attribute. The complete range need not be covered; anything not covered by any of the encoding ranges, and NULLs get the default value.

```
CREATE ENCODING <name> ON <user.table.attribute> AS
BEGIN
        (<lo|MIN>, <hi|MAX>, <id>)
        [, (<lo|MIN>, <hi|MAX>, <id>) ..]
        , <default_value>
END;
```

The above syntax creates a discretization for an attribute which can then be used in the GetRules command as follows:

```
GETRULES(T)..
...
USING <encoding> FOR <attribute>
```

An example of its usage follows (keywords are in uppercase). In the example, any value of age below 30 gets the identifier 1, values between 31 and 40 (inclusive) get the discrete identifier 2, and so on.

```
CREATE ENCODING e_age1 ON employees.Age AS
BEGIN
     (MIN, 30, 1),
     (31, 40, 2),
     (41, 50, 3),
     (51, 60, 4),
     (60, MAX, 5),
     ,0
END;

GETRULES(Emp)
INTO    EmpRB
WHERE   Body has { (Age=*) }
  and   Consequent is { (Car = 'BMW') }
  and   support > 0.1 and confidence > 0.4
USING   e_age1 FOR Age;
```

Encoding is a new object in the data dictionary and users can grant privileges on their own encodings to each other just like another database object. The optional **Using** clause in the GetRules forces the mining engine to apply the encoding *e_age1* when evaluating the above query.

Encoding an attribute automatically creates a corresponding decoding for it, which is employed by MSQL. The continuous columns displayed in rules thereafter show a `[lo, hi]` value pair, instead of the encoded value.

7. More examples

A few more examples are presented to further clarify the syntax, and illustrate some other capabilities of MSQL. The complete grammar for MSQL commands is presented in Virmani (1998). All examples below assume the same Emp schema described above. EmpRB is supposed to be the persistent rulebase mined with a minimum support of 0.1 and minimum confidence of 0.3.

Example 1. Select from the set of already mined rules, all rules with Nationality='Indian' in the Body and having up to 3 descriptors in the antecedent.

```
SelectRules(EmpRB)
where length <= 3
and ( Body has {(Nationality='Indian')}
```

Example 2. Find rules which subsume the above rule r in Body, have the same Consequent, and a higher confidence than r.

This is a classic example of mining around a rule, where we find an interesting rule and want to "expand" its Body (stronger condition) to gain confidence.

```
GetRules(Emp)
where Body has r.body
  and Consequent is r.consequent
  and confidence > r.confidence
```

Example 3. For every rule-body in R which has over 50 occurrences, generate a report showing the Body, and the associated number of rules it occured in.

This example demonstrates how reporting features of SQL can be employed to generate understandable summaries for Rulebases.

```
Project Body, count(*)
SelectRules(R)
group by Body
having count > 50
```

Example 4. Find rules in Emp of the form ABC→D, such that it has 10% more confidence than at least one of its immediate predecessors (e.g. AB→D).

This example uses table aliasing with rulebases and the SQL exists operator. It also presents an example of a nested GetRules query.

```
GetRules(Emp) r1
where length = 3
and exists ( GetRules(Emp) r2
             where length = 2
             and r1.Body has r2.body
             and r1.consequent is r2.consequent
             and r1.confidence > r2.confidence + 0.1 )
```

The main idea demonstrated in the above examples is that by restricting MSQL to a small subset, and by allowing full embedding of SQL within it, we can leverage the built-in data processing and reporting features of SQL. The representation of rules chosen, coupled with their persistent storage allows enormous post-processing capability with queries expressed in a few lines.

The next two sections describes the actual process of generating rules from the data. We first describe the evaluation procedure for what we defined as a "Basic-MSQL-query", and in the following section, present the evaluation of arbitrary MSQL queries by reducing them into the basic query form.

8. Basic query compilation

A Basic-GetRules query has similar syntax as the general GetRules query described above, except that it is a bit more restrictive. The "where" clause of the basic query may contain:

- conditions on Body
- conditions on Consequent
- conditions on support and confidence of rules required
- conditions on number of descriptors in Body (length conditions)
- sets of attributes that may not occur together in a rule (mutex conditions).

A Basic-GetRules query may **not** contain:

- disjunctions of expressions involving Body or Consequent,
- nested GetRules or SelectRules queries, or
- operations on partitioned tables, defined using "Create Partitioning" clause.

The last two conditions in the basic GetRules query (mutex and length conditions) simply alter the basic algorithm in a minor way and are discussed after presenting the basic algorithm. In terms of the rule patterns expected, the "where" clause of a simple rule query can be expressed in the following formulation, henceforth called a *D-condition*:

(MUST $\subseteq$ *Body) AND (Body* $\subseteq$ *MAY) AND (Consequent IN TARGET)*

The set *MUST* describes descriptors which *must* occur in rule bodies of the query answer[1], the set *MAY*, descriptors which *may* occur in rule bodies of the answer, and finally set *TARGET*, descriptors which *may* occur in rule consequents of the query answer (here only one descriptor from the *TARGET* may occur there). Notice that all three sets may contain, as a special case, all descriptors for a specific method. This would correspond to queries which, for example, ask about "all rules between method A and method B". A simple example of a GetRules query might be:

```
GetRules(C)
where Body has {(Age=[30,40])}
  and Body has {(Job='Manager') or (Job='Sales')}
  and support > 0.2
  and confidence > 0.3
```

For purposes of the algorithm, we consider discrete attributes only. Continuous attributes are supposed to be have been discretized off-line, using either user defined attributes, or the encode command described earlier. We will later describe generalization of this algorithm for an arbitrary disjunction of such conditions.

The overall algorithm follows an "eager evaluation" strategy, i.e. generate rules as soon as possible. We believe that it is more important to maintain a certain rate of rule production ("bandwidth") rather than wait for a long time and suddenly swamp the user with a thousand rules.

9. Algorithm

Given a query Q, the overall algorithm proceeds as follows:

1. First, identify the required methods in the database, then make a single pass through the database and build the necessary d-lists for MUST, MAY and TARGET. In this pass, methods that are not materialized in the database are also evaluated on an object-by-object basis.
2. Next, generate the Seed C-lists using the index-ANDing approach described above. If at *any* point during the formation, the C-list fails the support requirement, we exit.
3. Now recursively keep expanding each C-list in *Seed* with a d-list from descriptors in *MAY* until we either run out of support or construct all C-lists from descriptors in *MAY*. Each time a new C-list is generated proceed to *rule generation*, which can happen in either of the two ways:

    - *Rule Generation by Decomposition:* Given a C-list (C,L), take all d such that d is in C and d is in the intersection of *TARGET* and *MAY*, generate all rules (C-d, d), which meet support and confidence requirements imposed by the query.
    - *Rule Generation by Expansion:* Expand (not extend) C-list with each d-list for d in *TARGET – MAY* (but only one at a time!) and generate the rules (C,d) which meet support and confidence requirements of the query.

Reasons for providing two methods of rule generation are twofold: we want to do "eager evaluation" of rules, i.e. produce them as soon as possible, and we want to avoid the need for generating similar conjuncts repeatedly. The pseudo-code for the algorithm is presented in figure 1.

By significant extensions, we mean extensions which meet support requirements. The while loop in lines (5)–(10) will terminate either when we exhaust all extensions in *MAY*, or when there are no extensions possible, which can meet the support requirements. Note that while developing the Seed set for *MUST*, we can optimize by pruning the conjunctsets (and their future extensions) as soon as they fall below the minimum support. However, we still need to keep the ones that have too large support, since their extensions might fall into the acceptable range. Also important is the fact, that in line (9) we *extend* the seed rather than *expanding* it. This, coupled with the rule generation procedure, ensures that we get all rules, and that we don't produce any duplicates. Splitting rule generation into two different modules (lines 3 and 5) helps to avoid multiple generations of the same C-list. The main difference between these two steps is that C-lists contributing to the step (2) are further expanded, while C-lists in the step (5) are not used any more beyond the rule generation.

9.1. Introducing the mutex clause

It may often be the case that users might be interested in rules about certain methods present in a rule separately, but not together. An example could be methods like "county", and "phone area-code". It might be interesting to find rules across counties, and area codes,

```
main():                                                              (1)
  identify MUST, MAY and TARGET from the query                       (2)
  build all d-lists required for MUST, and the ones for MAY and TARGET (3)
  Seed := {all possible "significant" intersections of d-lists required in MUST}  (4)
  /* there could be more than one since we allow (method=*) constructs */
  while Seed ≠ ∅ {                                                   (5)
      s := get first element from Seed                               (6)
      Seed := Seed - {s}                                             (7)
      generate_all_rules(s)                                          (8)
      add all significant extensions of s obtained by merging s with
          d-lists from MAY to {Seed}                                 (9)
      }                                                              (10)

  generate_all_rules(S):                                             (1)
      let S = (C,L)                                                  (2)
      for each descriptor d ∈ { TARGET ∩ MAY ∩ C }                    (3)
          generate the rule (C-d,d) if it meets confidence & support (4)
      for each descriptor d ∈ {TARGET – MAY}                         (5)
          generate the rule (C,d) if it meets confidence & support   (6)
```

Figure 1. Pseudo-code for GetRules algorithm.

but perhaps not as interesting to find rules where counties predicted area-codes (county ⇒ area-code), or area-code and counties occured together in the Body.

The mutex primitive handles this situation by restricting the patterns which are allowed in the rules to be generated. Note that this construct in no way affects the expressive power of the language, it merely provides a cleaner way to specify and optimize the desired result. For example, the following query generates all rules with methods A,B,C,D in them, but where no two of A,B,or C occur together.

```
GetRules(C)
where Body in {A,B,C,D}
  and Consequent in {A,B,C,D}
  and mutex(A, B, C)
  and support > 0.2
```

Expressing this without using this primitive leads to a very complicated disjunctive expression, which is harder to optimize.

The presence of the mutex expression alters the algorithm in the following way: A two dimensional table is created for all attributes involved in the mutex clauses present in the query. A conjunctset C, during expansion is checked for the presence of one of the attributes from the mutex set, and if present, all pairwise mutually exclusive attributes to it are not used in its expansion. Since this is done starting with conjunctsets of length 1, it is assured that a conjunctset will never have more than 1 method from the mutex set. For each extra table lookup for a method, this structure saves one merge operation.

9.2. Introducing the length clause

The length construct was added to the basic query specification to permit users to easily specify queries like "Show me rules with at least three descriptors in Body", or "Find all rules between given attributes at most k-descriptors in Body". The handling of this construct can be divided into three cases:

- length $<$ k. In this case, a global variable min_length keeps the minimum length of a conjunctset in the seed list. At each iteration, conjunctsets in MAY which are of length greater than k- min_length are deleted. Also, while merging a conjunctset from the seed set with another in the MAY set, if the resulting conjunctset is bigger than size k, the merge is not performed.
- length $>$ k. In this case, rule generation steps are skipped from the algorithm, until the seed set has one or more conjunctsets of size k+1. From then on, we start rule generation for cases, where the seed conjunctset has a size larger than k.
- length $=$ k. This is a combination of the above two cases, and is handled by combining them.

9.3. Example

This section presents a complete walk-through of the algorithm on a simple dataset to aid in understanding of the various steps. The example query is shown below:

```
GetRules(T) R
where R.Body in {(Disease=*), (Age=*), (Occupation = *)}
      and R.Body has {(Disease = Pneumonia),(Age = Middle)}
      and R.Consequent in {(ClaimAmt=*), (Occupation =*)}
      and Support > 0.07
      and Confidence > 0.35
```

We assume that both Age and ClaimAmt methods have been discretized already by partitioning the values of the corresponding materialized attributes and have three values each: *Young*, *MiddleAge* and *Old* for Age and *Low*, *Medium* and *High* for the ClaimAmt method.

Comparing with the standard form of D-CONDITION, we have in the above query:

MUST = {(Disease = Pneumonia),(Age = Middle)}
MAY = {(Disease=*), (Age=*), (Occupation = *)}, and
TARGET = {(ClaimAmt=*), (Occupation = *)}

We use the "*" symbol to denote all descriptors for a particular method.

We will assume the method order to be Disease, Age, Occupation, and ClaimAmt for this example. Following the pseudocode, we consider only those objects in the database which have (Disease = Pneumonia) and (Age = Middle). For this subset of the database,

Table 1. d-lists for the example dataset.

(Disease=Pneumonia)	3,7,11,15,17,24,26,33,37
(Age=Middle)	3,7,11,15,17,24,26,33,37
(Occupation=Clerk)	11,15,37
(Occupation=Fieldworker)	7,17,24,33
(Occupation=Nurse)	3,26
(ClaimAmt=Low)	3,7
(ClaimAmt=Medium)	11,17,24
(ClaimAmt=High)	15,26,33,37

we generate d-lists for the four methods needed for this query. For the sake of illustration, let Table 1 show all the d-lists generated for this dataset of size, say, 40 objects.

To generate the initial seed, we intersect the d-lists corresponding to (Disease = Pneumonia) and (Age = Middle). In this example, this is the only element in the Seed set, and it meets the minimum support of 3 objects (7% of 40).

Before we extend the seed, we first generate all possible rules with it. In this case, we can only get rules by expansion, since $\{C \cap TARGET \cap MAY\} = \emptyset$. Expanding the seed with ClaimAmt (only method in (TARGET – MAY)), we get the following rule which meets the support and confidence constraints:

```
(Disease=Pneumonia),(Age=Middle) => (ClaimAmt=High)  [s:9, c:44%]
```

The other two expansions fail the confidence requirement of 35%.

Next, since Occupation is the only method in *MAY* which does not occur in *MUST* we add to our *Seed* set, d-lists of the form (Occupation = $*$). That gives us two C-lists meeting the support requirement:

```
(Disease=Pneumonia),(Age=Middle),(Occupation=Clerk),  {11,15,37}
(Disease=Pneumonia),(Age=Middle),(Occupation=Fieldworker),
  {7,17,24,33}
```

For the above seeds, both decomposition, and expansion methods for rule generation can be applied. The first of above C-lists forms the following rules:

```
(Disease=Pneumonia),(Age=Middle) => (Occupation=Clerk)
  [s:9, c:33%]
(Disease=Pneumonia),(Age=Middle),(Occupation=Clerk)
  => (ClaimAmt=High)  [s:3, c:67%]
```

the first of which fails the confidence requirement, and is rejected. The second C-list yields:

```
(Disease=Pneumonia),(Age=Middle) => (Occupation=Fieldworker)
[s:9, c:44%]
(Disease=Pneumonia),(Age=Middle),(Occupation=Fieldworker) =>
(ClaimAmt=Medium)   [s:4, c:50%]
```

both of which are acceptable rules.

10. General query compilation

The syntax of the most general GetRules query was described in Section 4. Earlier, we described the evaluation of what we defined as the "Basic-MSQL-query". In this section, we describe the evaluation procedure for general MSQL queries.

Evaluation of the general query is presented as follows. First, we consider each of the extensions to the Basic-MSQL-query individually, and using some pre- and post- processing, reduce them into the "Basic-MSQL-query" formulation discussed in the last section. Next, in Section 12, we present the pseudocode for the overall query processing algorithm which takes a general query with all such extensions together, and transforms them to the simplest MSQL query form in a well defined manner. Finally, we draw some observations from the algorithm, and present some optimizations to it.

11. Reduction of nested subqueries

Based on the evaluation model, nested queries can be classified into two basic types:

- *Stratified queries*, in which the subquery can be evaluated fully before the outer query needs to be considered, and
- *Correlated queries*, in which the subquery is connected to the outer query via correlated variables, and the execution plan requires some form of "loop unfolding".

Both these query types are discussed below.

11.1. Stratified subqueries

The subquery condition in this case looks like one of the following:

```
{Body|Consequent} IN <nested Q'>} |
{confidence|support} {<|>|=|<=|>=|<>} <nested Q'> |
[NOT] EXISTS {<nested Q'>}
```

In the first type of condition, nested-Q' is a query that returns a set of objects of the same type as the object used in the left side of the IN operator. In the second case, the query returns a single row containing a value which yields a predicate on confidence or support. In the third condition type, the nested-Q' evaluates to some finite set of values (which yields either True or False in the presence of EXISTS/NOT EXISTS).

The evaluation method of such queries is "bottom-up". The subquery is evaluated first, and is replaced with its results in the outer query. The outer query is now free of this stratified query. This process is applied to each stratified subquery till the outer query is reduced to the simple MSQL query type.

The pseudocode for stratified evaluation of nested queries can be expressed as:

EvaluateQ(Q**)** (1)
begin (2)
Let Q contain a stratified subquery clause SSQ (3)
if SSQ is of the form: *expr IN nested-Q* then (4)
Let tmpResult = EvaluateQ(*nested-Q*) be $< v_1, v_2..v_n >$ (5)
Qnew = Replace SSQ with $(expr = v_1) \vee .. \vee (expr = v_n)$ (6)
else if SSQ is of the form: *[NOT] EXISTS nested-Q* then (7)
tmpResult = EvaluateQ(*nested-Q*) (8)
Qnew = Replace SSQ with "[*NOT*](*tmpResult* $\neq \phi$)" (9)
end if; (11)
EvaluateQ(Qnew) (12)
end (13)

11.2. Correlated subqueries

The subquery format for these queries has similar syntax as for stratified queries except that the inner query contains references to one or more instances of classes (or methods within those classes) from the outer query. This makes it impossible to evaluate either the outer query or the inner query independently of the other.

For purposes of query evaluation, we define **a correlated variable** as a reference in the inner (nested) subquery, to a class or a method within a class which is being computed in the outer query. For example, the following query generates those rules above 10% support and 30% confidence, which have no successors.

```
Project Body, Consequent, support, confidence
GetRules(C) as R1
where support > 0.1 and confidence > 0.3
  and not exists ( GetRules(C) as R2
                   where support > 0.1 and confidence > 0.3
                   and R2.Body has R1.Body
                   and not (R2.Body is R1.Body)
                   and R2.consequent is R1.consequent )
```

In the above query, references to R1.Body and R1.consequent in the subquery are two instances of correlated variables. Furthermore, let us define an *instantiation* of a correlated subquery as follows: Let $Q(R_1, R_2..R_n)$ be a query over classes $C_1..C_n$, such that R_i corresponds to the Rule class generated from C_i. Let $CSQ(vR_1, vR_2..vR_k)$ be the correlated subquery within Q, containing correlated variables $vR_1..vR_k$, which are references to rule

classes $R_1..R_n$ defined outside the scope of CSQ. Then an **instantiation** or "grounding" $I_{r_1,r_2..r_n}(CSQ)$ of the subquery is defined as a complete substitution of the correlated variables in CSQ, such that r_i is an element of R_i, and

- if vR_i is a reference to method M of class R_i, it is replaced with $r_i.M$, value of method M in r_i.
- if vR_i is a reference to the whole class R_i, it is replaced with the instance r_i of R_i.

The same definition applies to stand-alone "where" clause expressions within a query. The instantiation, as defined above, is used in the evaluation of correlated subquery conditions. Contrary to stratified queries, where the outer and inner queries are evaluated separately, correlated queries can be evaluated either top-down or bottom-up using "loop unfolding". Both these ideas are described in the subsections below.

11.3. Top-down evaluation

The idea in top-down evaluation follows the "generate and test" strategy—we evaluate the outer query first while ignoring the correlated subquery condition, in effect generating a superset of the answers. Then for each resulting row, we test if the subquery holds. The procedure is described more formally below.

Let $Q(R_1, R_2, ..R_n)$ be an MSQL query, containing a subquery $CSQ(vR_1, vR_2..vR_k)$, where vR_i is a correlated reference to R_i, as defined above. The "where" clause conditions of Q can then be written as: $COND \wedge CSQ(vR_1, vR_2, ..vR_k)$. Here is how we evaluate this query:

1. Let Q′ be the more general query formed by substituting CSQ with "True" in Q. In other words, the answer A of Q is now a subset of the answer A' of Q'.
2. Evaluate Q′, without applying any projections yet. The answer A' of Q' is then the set of tuples $\langle r_1, r_2..r_n\rangle \subseteq \{\times_{i=1}^{n} R_i\}$
3. The answer A to the original query Q is then the set of those tuples $\langle r_1, r_2..r_n\rangle$ (with projections now applied), for which the instantiation $I_{r_1,r_2..r_n}(CSQ) = true$

11.3.1. Example: To illustrate the above procedure, let us apply it to the example query presented above. After eliminating the correlated subquery from this query, we get:

```
Project Body, Consequent, support, confidence
GetRules(C) as C1
where support > 0.1 and confidence > 0.3
```

Let us say the results of the above query are as follows:

```
(A=a1) ==> (C=c1)   [0.33, 0.40]
(A=a1) (B=b1) ==> (C=c1) [0.15, 0.35]
(B=b2) (C=c3) ==> (D=d5) [0.11, 0.60]
```

The respective instantiations of the subquery, formed using the above result set are:

```
not exists ( GetRules(C) as R2
     where support > 0.1 and confidence > 0.3
     and R2.Body has { (A=a1) }
             and not (R2.Body is { (A=a1) })
         and R2.consequent is { (C=c1) } )

not exists ( GetRules(C) as R2
     where support > 0.1 and confidence > 0.3
     and R2.Body has { (A=a1), (B=b1) }
             and not (R2.Body is { (A=a1), (B=b1) })
         and R2.consequent is { (C=c1) } )

not exists ( GetRules(C) as R2
     where support > 0.1 and confidence > 0.3
     and R2.Body has { (B=b2), (C=c3) }
     and not (R2.Body is { (B=b2), (C=c3) })
         and R2.consequent is { (D=d5) } )
```

On evaluation of these queries, we find that the second and third one are true, which means that the last two rules belong to the set of valid results. Applying the projection operation on them doesn't alter anything in this case, and so our final result is:

```
(A=a1) (B=b1) ==> (C=c1) [0.15, 0.35]
(B=b2) (C=c3) ==> (D=d5) [0.11, 0.60]
```

11.4. Bottom-up evaluation

Contrary to the top down evaluation, the bottom-up evaluation attempts to unfold the inner query into a large boolean condition formed by evaluating *CSQ* without the correlating conditions first, and then instantiating the correlating conditions with the answers thus obtained.

Let $Q(R_1, R_2, ..R_n)$ be an MSQL query, containing a subquery $CSQ(L_1, L_2..L_j, vR_1, vR_2..vR_k)$, where vR_i's are correlated references to R_i outside *CSQ*'s scope, and L_i's are the rule classes corresponding to classes within *CSQ*. The steps to the algorithm can then be expressed as follows:

1. Let the "where" clause of *CSQ* be of the form $(CC \wedge Rest)$, where CC is the set of conditions involving correlated expressions. Let CSQ' be the query formed by removing CC from the "where" clause in *CSQ*.
2. Evaluate CSQ'. The result is a set of tuples A of the form $\langle r_1 \ldots r_j \rangle$ such that $r_i \in L_i$ and $\langle r_1 \ldots r_j \rangle \subseteq \{\times_{i=1}^{n} L_i\}$
3. For each tuple $t = \langle r_1 \ldots r_j \rangle$ in the intermediate result, create an instantiation INS_t formed by instantiating CC with t. In other words, $INS_t = I_{\langle r_1 \ldots r_j \rangle}(CC)$.

4. Replace *CSQ* in *Q* with the expression $\bigvee_{t \in A}(INS_t)$.

What we have effectively done in the above procedure is to "flatten" the nested query into a large logical OR expression made from intermediate results of the subquery.

To help draw a comparison later, we apply this procedure on the same example as in the above section.

11.4.1. Example: In the correlated subquery example presented above, the correlated subquery looks like:

```
( GetRules(C) as R2
   where support > 0.1 and confidence > 0.3
   and R2.Body has R1.Body
   and not (R2.Body is R1.Body)
   and R2.consequent is R1.consequent )
```

After removing the "where" clause conditions which have references to correlated variable R1, we are simply left with:

```
GetRules(C) as R2
   where support > 0.1 and confidence > 0.3
```

Let us say the results of the above query are as follows:

```
(A=a1) ==> (C=c1)         [0.33, 0.40]
(A=a1) (B=b1) ==> (C=c1) [0.15, 0.35]
(B=b2) (C=c3) ==> (D=d5) [0.11, 0.60]
```

The respective instantiations $I_1..I_3$ of the correlated conditions in the subquery, formed using the above result set are:

```
I1:              { (A=a1) } has R1.Body
                 and not ({ (A=a1) } is R1.Body)
                 and { (C=c1) } ) is R1.consequent

I2:              { (A=a1), (B=b1) } has R1.Body
                 and not ({ (A=a1), (B=b1) } is R1.Body)
                 and { (C=c1) } is R1.consequent )

I3:              { (B=b2), (C=c3) } has R1.Body
                 and not ({ (B=b2), (C=c3) } is R1.Body)
                 and { (D=d5) } is R1.consequent
```

The correlated subquery in the outer query can now be removed and replaced with the logical OR of boolean conditions $I_1..I_3$. The outer query is now free of correlated subqueries and looks like:

```
Project Body, Consequent, support, confidence
GetRules(C) as R1
where support > 0.1 and confidence > 0.3
  and not exists ( (I1) or (I2) or (I3) )
```

On evaluation of the above query, we find that the second and third rules produced above satisfy the boolean conditions, and thus are part of the overall result set. Applying the projection operation on them doesn't alter anything in this case, and so our final result is:

```
(A=a1) (B=b1) ==> (C=c1) [0.15, 0.35]
(B=b2) (C=c3) ==> (D=d5) [0.11, 0.60]
```

11.5. Deciding between top-down and bottom-up evaluation

For correlated queries, both top-down and bottom-up evaluation procedures perform in spirit, what is commonly referred to as "loop unfolding" in compilers research. Irrespective of whether the outer or the inner query is "flattened", both transformations lead to the same result. However, for certain types of queries, it may be much more efficient to choose one method over the other. In this section, we first discuss two differentiating examples, and then generalize the observations from these into a set of heuristics, similar to the ones used in database optimization, to be used by the general evaluation algorithm.

Consider the following example, which tries to generate all rules such that a successor for the rule exists with a support above 30 percent:

```
Ex1:    GetRules(C) as R1
        where exists (  GetRules(C) as R2
                        where R2.Body has R1.Body
                          and R2.length > R1.length
                          and R2.Consequent = R1.Consequent
                          and R2.support > 0.3 and R2.confidence
                            > 0.4
                     )
```

Consider a second example, in which the query returns all rules above 40 percent support and 50 percent confidence which have an immediate predecessor within 20 percent of the rule's support.

```
Ex2:    GetRules(C) as R1
Where support > 0.4 and confidence > 0.5
          and exists (  GetRules(C) as R2
                        where R2.Body in R1.Body
                          and R2.length = R1.length-1
                          and R2.Consequent = R1.Consequent
                          and R2.support < 1.2*R1.support
                     )
```

In the first example, the only constraints on rule generation are present in the inner query. The outer query is almost "unsafe" in a way, since the universe of rules is not present "extensionally" in a table, as is the case with similar database queries. Expanding the outer query first would almost be futile, as the number of rule combinations to generate and test will be truly exponential. However once the inner query is unfolded, we have a finite number of "successor" rules in R2 for which we have to look for predecessors.

In the second example, the inner query does have constraints on support and other attributes, but all the constraints are part of "correlated expressions" as defined earlier. In order to follow bottom up evaluation, we will have to remove them first, and then execute the remaining query, which will lead to a similar problem as in the first one. In this case, it is easy to see that the outer query should be given preference.

In general, the optimized algorithm must first examine the conditions *independent of correlated expressions* in both inner and outer query, and then select the query which has more restrictive conditions.

It is not possible to determine which query will generate fewer rules, but we have found the following heuristics to work well in practice:

1. The query with higher support threshold should be evaluated first.
2. Failing support, confidence should be considered next.
3. A $(length = x)$ type expression is in general, more restrictive than a $(length > x)$ expression, which is in general, more restrictive than a $(length < x)$ expression.
4. A "Body IS (constant expression)" should be given higher priority over a "Body HAS" expression, which should be preferred over a "Body IN" type of expression.
5. "Consequent IN" expression.
6. 6) When comparing descriptors, a descriptor of the form $(A = a_i)$ is much more restrictive than a wild-card descriptor of the form $(A = *)$.

The fourth heuristic above may not always lead to the best choice. For example, an expression like "*Body HAS* $(A = *)$" might turn out to be less restrictive than "*Body IN* $(C = c_3)$, $(B = b_4)$, $(D = d5)$", which must examine all subsets of the three descriptors. But if the two expressions are present in the outer and inner queries, they both must eventually be evaluated by the query. Besides, once it boils down to selecting the more "tighter" constant, it doesn't make a crucial difference in performance. The heuristics are there just to help prevent selecting an almost unconstrained query from being expanded.

There still exist several pathological cases, where no heuristic will work. Consider the following example, which generates all rules of the form $(X = x_i) \Rightarrow (Y = y_i)$ such that the reverse rule $(Y = y_i) \Rightarrow (X = x_i)$ also exists with support and confidence within 5 percent of the original rule.

```
GetRules(C) as R1
where length = 1
  and exists (GetRules(C) as R2
              where length = 1
                and R1.Body = R2.Consequent
                and R2.Body = R1.Consequent
```

```
                and R2.support >= 0.95 * R1.support
                and R2.support <= 1.05 * R1.support
                and R2.confidence >= 0.95 * R1.confidence
                and R2.confidence <= 1.05 * R1.confidence)
```

The only non-correlated condition in both outer and inner queries is (length = 1). In either type of evaluation, the evaluation procedure must consider every element of the set $\{R \times R\}$, where R is the set of rules of length 1 in C, and validate the rest of the conditions.

Before applying the heuristics, however, one must propagate constants (as in database query evaluation) between inner and outer queries whenever possible. This is usually the case when either the inner or the outer query contains a comparison of an expression with a constant, and the other query compares the same expression with the corresponding expression in its own scope. For e.g., in the example below, the inner query can *additionally* have the condition "($length > 3$)" without losing any results.[2]

```
...
where R1.length > 3
  and exists (..
                where R2.length > R1.length)
```

12. Overall evaluation algorithm

We can now combine the procedures developed in Sections 11.1, 11.4 and 11.3 to take an arbitrary query with subqueries, and transform it into a query without subqueries. (Note: the assumption still in use is that the all query conditions are "AND"-ed together. Disjunctions are dealt with in the following section).

The overall query evaluation function "EvaluateQ" can be described by the following pseudocode. It recursively calls EvaluateQ to remove flatten subquery conditions, and finally calls the Basic-MSQL-Eval algorithm described earlier.

EvaluateQ(Q**)**
begin
    Let Conds be the "where" clause conditions in Q
    if Conds contains a stratified subquery SSQ /* See definition */
        Reduce SSQ to a set of boolean conditions as in 11.1
        Let Q' be the query thus formed. EvaluateQ(Q').
    else if Conds contains a correlated subquery CSQ /* See definition */
        Examine the conditions in Q and CSQ, and using heuristics developed
        in 11.5, decide between top-down and bottom-up unfolding.
        if top-down, then
            Use EvaluateQ in the procedure described in 11.3 to
                reduce the query to a set of subquery instantiations.
            Apply EvaluateQ on each instantiation recursively and union the results.
        else if bottom-up, then

```
            Flatten the inner query to a boolean condition as described in 11.4
            Let Q' be the resulting query.
            EvaluateQ(Q′).
        end if
    else Basic-MSQL-Eval(Q)
end
```

13. Query evaluation in the presence of disjunctions

Until now, we assumed that our query conditions were composed using only the AND operator. In the more general case, a query can have multiple conditions connected using a combination of the "AND" and the "OR" operators. In a naive evaluation scheme, one can reduce a general boolean condition correctly into a union/intersection of the results of queries over each individual condition as shown below:

```
Evaluate_SimpleQ(P, T[A1..An], Conds)                                  (1)
begin                                                                  (2)
    if C is of the form (C1 OR C2) then                                (3)
        Result = (Evaluate_SimpleQ(P, T[A1..An], C1)                   (4)
                  ⋃ Evaluate_SimpleQ(P, T[A1..An], C2))
    else if C is of the form (C1 AND C2) then                          (5)
        Result = (Evaluate_SimpleQ(P, T[A1..An], C1)                   (6)
                  ⋂ Evaluate_SimpleQ(P, T[A1..An], C2))
    else                                                               (7)
        Result = Basic_MSQL_Eval(P, T[A1..An], C)                      (8)
    end if;                                                            (9)
    return Result                                                      (10)
end;
```

Although correct, the above procedure will result in extremely inefficient evaluation. For example, in a rule condition like:

```
Body has { A, B } and support > 0.5 and consequent is { F }
```

it will process three queries, each with a single component condition above and then take intersection of the results. This will lead to generation of all rules with support >0.5, all rules with Consequent $=\{F\}$, and all rules containing A and B in the Body, and then intersecting the results. The Basic_MSQL_Eval procedure can already accept queries with multiple conditions ANDed together.

In case of a disjunction, evaluating the query by first evaluating each operand of the OR, and then forming set theoretic union will in general result in replication of the same work. As an example, when *MUST* sets for different disjunctions intersect, the intermediate C-lists

obtained in the construction of a C-list for one atom may be reused for another. Consider the following motivating example:

```
GetRules(C)
where (Body in {(Age=*),(Sex=*),(Salary=*)}
       or Body in {(Age=*),(Job=*),(Salary=*))
  and Consequent in {(Car=*)}
  and confidence > 0.4
  and support > 0.1
```

The intermediate C-lists for (Age=*)(Salary=*) produced for the first Body expression could be recycled for the second. However, stand-alone evaluation of both clauses will produce, as its result, conjunctsets of the form "(Age=v1)(Sex=v2)(Salary=v3)", which cannot be used for generating conjunctsets in the second Body expression.

The optimized algorithm for evaluating disjunctive queries uses the GetRules evaluation engine described earlier as a procedure, and makes multiple calls to it. However, it is different from the Evaluate_SimpleQ procedure in its approach. There are four key issues that we have identified in the evaluation of disjunctive queries:

- Distribution pruning conditions over disjuncts.
- Reusing conjunctsets between disjuncts.
- Evaluation order of disjuncts.
- Swapping Priority of conjunctsets.

Below we discuss each issue separately, and finally present our algorithm.

13.1. Distributing the constraints:

An important observation is that support, length and mutex conditions happen to be the only pruning conditions that can control the number of possible conjunctsets generated. The idea therefore is to distribute the pruning conditions over the OR expressions as much as possible, and invoke the **Basic-MSQL-eval** engine with as many constraints from logically "AND-ed" subexpressions as possible, rather than making multiple invocations. For instance, a query like

```
(Body has {A,B} or Body has {C,D})
                    and support > 0.5 and consequent is {F}
```

should result in two invocations like:

```
Basic-MSQL-Eval(Body has {A,B} and support > 0.5 and consequent
  is {F})
Union
Basic-MSQL-Eval(Body has {C,D} and support > 0.5 and consequent
  is {F})
```

as opposed to:

```
Basic-MSQL-Eval(support > 0.5)
Intersect
Basic-MSQL-Eval(consequent is {F})
Intersect
( Basic-MSQL-Eval(Body has {A, B})
  Union
  Basic-MSQL-Eval(Body has {C, D}) )
```

13.2. Reusing conjunctsets between invocations

The second task of the algorithm should be to identify common subexpressions between two calls to the Basic-MSQL-eval engine, and prevent their regeneration *from the disk* a second time.

To formalize this notion of reusing work between two disjuncts, let us first define a *conjunct-pattern* to be the expression $A_1A_2..A_n$ representing the family of conjunctsets of the form "$(A_1 = a_{1i})(A_2 = a_{2j})..(A_n = a_{nk})$", where A_i's are the methods of the class to be mined. Instantiations of conjunct-patterns yield C-lists, and are said to be "generated" by the respective pattern. For instance, a conjunct-pattern $(A = *)(B = *)$ generates C-lists of the form $((A = a_1)(B = b_1), L)$, and $((A = a_2)(B = b_2), M)$, and so on, where a_i's and b_i's belong to the domains of A and B respectively, and L and M are the support lists for the respective conjunctsets.

Consider two disjuncts $D1$ and $D2$. Let the set of C-lists generated by a conjunct-pattern P for $D1$ be $P(D1)$, and for $D2$ be $P(D2)$. We then say that $D1$ subsumes $D2$ under P, iff the set $P(D1)$ is a superset of $P(D2)$ for any dataset. This is expressed as:

$$D1 \xrightarrow{P} D2 \; iff \; P(D1) \supseteq P(D2)$$

The set of reusable conjunct-patterns from $D1$ to $D2$ is then the set of conjunct-patterns P such that $D1$ subsumes $D2$ under P. This can be expressed as:

$$ReUse(D1, D2) = \{ P \mid P(D1) \supseteq P(D2)\}$$

The intuition behind using a conjunct-pattern as opposed to a C-list to define the granularity of reusable work between two disjuncts is as follows. Let $D1$ and $D2$ share C-lists generated by a pattern P, and let $\{i_1, i_2, i_3\}$ be the instantiations of P needed in $D1$, and let $\{i_1, i_2, i_3, i_4, i_5\}$ be the corresponding set for $D2$. To generate these C-lists, the algorithm must scan the database to construct the support list for each instantiated conjunctset. If we were to reuse the C-lists $i_1..i_3$ from $D1$ to $D2$, the algorithm would still have to scan the database for generating the remaining C-lists i_4 and i_5. This does not result in much savings, since the algorithm doesn't know how many more instantiations of P could meet the conditions in $D2$, and must "count" each possible instantiation of P during the data scan. However, if we can *guarantee* that the set of instances for a pattern in $D2$ is a superset

of the instances needed in $D1$, then the algorithm need only "refine" the existing set by pruning it further, without the need to go to the data (on disk) a second time. Therefore, even though the first three C-lists can be reused in either direction in the example above, it is the reusability of the pattern P as a whole (which is only possible from $D2$ to $D1$ in this case) which promises some savings for the algorithm. Note that the definition of *ReUse* does not involve a given data set, hence we are interested in conjunct-patterns that can be claimed to be reusable across two disjuncts, for any general data set.

Below, we describe a way to compute the ReUse sets between several disjuncts of a query, when each disjunct contains only logically "AND"-ed conditions on Body, Consequent, support, confidence and mutex. Further, the conditions on Body and Consequent within each disjunct are assumed to have been reduced to the form:

(MUST $\subseteq$ Body) AND (Body $\subseteq$ MAY) AND (Consequent IN TARGET)

as described for the Basic evaluation algorithm.

We now define the terms *conj_reuse* and *descrip_reuse* between two disjuncts D_1 and D_2, and later use them to derive *ReUse*($D1$, $D2$). Intuitively, *conj_reuse*($D1$, $D2$) returns the conjunct-patterns which can be reused from $D1$ to $D2$, if possible, and *descrip_reuse* ($D1$, $D2$) tries to capture descriptor patterns which can be reused from $D1$ to $D2$.

Let DQ be the disjunctive query, with the "where" clause of the form $(D_1 \vee D_2 \vee .. \vee D_n)$, where D_i is a disjunct of the form mentioned above, and let $MUST_i$, MAY_i and $TARGET_i$ be the corresponding MUST, MAY and TARGET sets for disjunct D_i.

Let $MUST_i$ and $MUST_j$ be the the $MUST$ sets for D_i and D_j, then the **conj_reuse**, between D_i and D_j can be computed as follows:

conj_reuse(D_1, D_2**):**

```
    if length(MUST_1) <= 1, return φ;                          (1)
    if min_support in D_1 > min_support in D_2 return φ;       (2)
    if (signature(MUST_1) = signature(MUST_2) and
        (MUST_1 has MUST_2)) then                              (3)
            Let {A_1, ...A_n} be the attributes used in MUST_1. (4)
            Return the conjunct-pattern "A_1 A_2..A_n".        (5)
    else return φ;
```

Line (1) above returns the empty set if either of the MUST sets is a single descriptor. In this case, the MUST-descriptor is used in the next procedure descrip_reuse. The rest of the code simply verifies that the signatures match and the MUST pattern of the first disjunct is more general than the second.

As computed by the above procedure, conj_reuse between must sets `(A=a2)(B=*)` and `(A=a2)` is an empty set, since their signatures don't match, and conj_reuse between `(A=a2)(B=*)` and `(A=*)(B=*)` is empty, since the first doesn't contain the second, while conj_reuse between `(A=*)(B=*)(C=c3)` and `(A=a2)(B=*)(C=c3)` is the conjunct-pattern (A=*)(B=*)(C=*).

The "strictness" of conj_reuse follows from the following observation: The MUST set represents the descriptors that must all be present in the Body, hence the GetRules evaluation

engine directly pushes these conditions on the data and "counts" only those records which have all conditions in them. Therefore the algorithm will build the initial C-lists quite differently for a query requiring (A=∗)(B=∗) as its MUST set, versus for a query requiring (A=a1)(B=∗)(C=c3) as the MUST set, even though the condition (A=a1)(B=∗) is common to both of them. On the other hand, for the *TARGET* and *MAY* sets, the algorithm constructs singleton d-lists from the data, which are the simplest patterns and thus easier to recycle, as opposed to conjunct-patterns, since the more complicated the pattern, the less the probability of another disjunct needing it.

Let MAY_i, $TARGET_i$ and MAY_j, $TARGET_j$ be the the *MAY* and *TARGET* sets for disjuncts D_i and D_j. Let MT_i represent the union of descriptors from MAY_i and $TARGET_i$, and also $MUST_i$, if length($MUST_i$) = 1 (when they are not handled by conj_reuse), with the following transformation applied: All instances of descriptors $(A = a_1)$, $(A = a_2)$, .. $(A = a_n)$, sharing the same attribute are collected together into one expression of the form $(A = \{a_1, a_2, .., a_n\})$. Thus MT_i contains just one descriptor per attribute.

We can now compute the descrip_reuse between D_i and D_j, as follows:

descrip_reuse(D_1, D_2):
if min_support in D_1 > min_support in D_2 return ϕ; (1)
for every attribute A_i such that descriptor $(A_i = V_1) \in MT_1$
and $(A_i = V_2) \in MT_2$ { (2)
If (V_1 = "*" || $V_1 \supseteq V_2$), add $(A_i = *)$ to result (3)
}
return result (4)

For example, if MAY_1 is {(A=∗), (B=b1), (C=∗)} and $TARGET_1$ is {(F=∗)}, while MAY_2 is {(A=∗), (F=∗)} and $TARGET_2$ is {(C=C3), (B=∗)} then *descrip_reuse*(D_1, D_2) is the set {(A=∗), (C=∗), (F=∗)}.

Claim: The set *ReUse*(D_i, D_j) can be computed as:

$$ReUse(D_i, D_j) = conj\_reuse(D_i, D_j) \cup descrip\_reuse(D_i, D_j)$$

Proof: *descrip_reuse* captures individual descriptor patterns which can be reused from D_i to D_j, which *conj_reuse* captures longer length patterns which can be reused between the same disjuncts.

Let us first assume **that** the descriptor pattern $(A = *) \in ReUse(D_i, D_j)$. This implies that both D_i and D_j compute some subset of **all** descriptors of the form $(A = a_i)$. In addition, **it** implies that the set of such descriptors generated in D_i is guaranteed to be a superset of the set of such descriptors computed in D_j, for any database. This guarantee is only possible if the minimum support requirement in D_i is less than the similar requirement in D_j, and the query pattern computing descriptors of the form $(A = a_i)$ is either unrestricted $((A = *))$, or is more general (enumerates more instances) than its counterpart in D_j. What we listed above through semantic reasoning are exactly the procedural steps to compute *descrip_reuse*(D_i, D_j).

Now assume that the conjunct-pattern $(A = *)(B = *)..(N = *) \in ReUse(D_i, D_j)$. Following similar reasoning as above, we can arrive at the conclusion that the above conjunct-pattern must be computed by *conj_reuse*(D_i, D_j).

Since *conj_reuse* captures all conjunct-patterns of length > 1, which fit the definition of *ReUse*, and *descrip_reuse* captures all descriptors (conjunctsets of length 1) which fit the above definition, anything that is in *ReUse* is computed by the right hand side. Anything computed by the right hand side, by definition is in *ReUse*, since the procedures implement the definition of *ReUse*.

Claim: if the sets $ReUse(D_i, D_j)$ and $ReUse(D_j, D_i)$ are both non-empty, the minimum support required in disjuncts D_i and D_j is the same.

Proof: For $ReUse(D_i, D_j)$ to be non-empty, it means that there exists some conjunctset C for which all patterns computed by disjunct D_i are a superset of similar patterns computer by D_2, irrespective of the database. This is possible only if support$(D_1) \geq$ support(D_2). Similarly for $ReUse(D_j, D_i)$ to be non-empty, it must follow that support$(D_j) \geq$ support(D_i). Together, these two cases imply that support$(D_i) =$ support(D_j).

13.3. Conjunctset caching and replacement policies

Empirical studies of the rule generation process, including the results presented in Virmani (1998), show that the CPU time is a small fraction of the elapsed time, indicating the I/O bound nature of the problem. One of the main reasons for high disk activity is that the available memory cannot hold all the intermediate conjunct-patterns needed by less-restrictive queries. Given infinite memory, one can trivially generate and maintain the union of all d-lists required by all disjuncts for the duration of the query, avoiding the need for regeneration or swapping. However, given a finite memory, it becomes important for the algorithm to ensure minimal disk activity and maximum reuse of conjunctsets.

More specifically, once the pairwise reuse sets are known, the algorithm must examine the following two issues:

- Given a graph where each vertex is a disjunct, and directed edges from disjunct D1 to D2 represent $ReUse(D_1, D_2)$, come up with a "traversal" of this graph to ensure maximum conjunctset reuse.
- When faced with a decision to swap certain conjunct-patterns out of memory in order to make progress, come up with a strategy to minimize the number of such swaps.

As we shall see later, several well established heuristics from operating systems field can be applied here to guarantee best performance for disjunctive query evaluation.

The graph we arrive at, by following the construction mentioned above, has the form shown in figure 2. The disjuncts can be partitioned into levels of increasing minimum support, with disjuncts sharing the same minimum support level belonging to the same level. Between two levels, we are guaranteed to have edges only going from lower to higher support levels. This implicitly gives us a partial order on disjunct evaluation. In addition, we label each reuse-edge with the number of conjunct-patterns that are being reused from source to destination. This serves us a weight assignment for each edge, later used in a heuristic to break ties, if necessary.

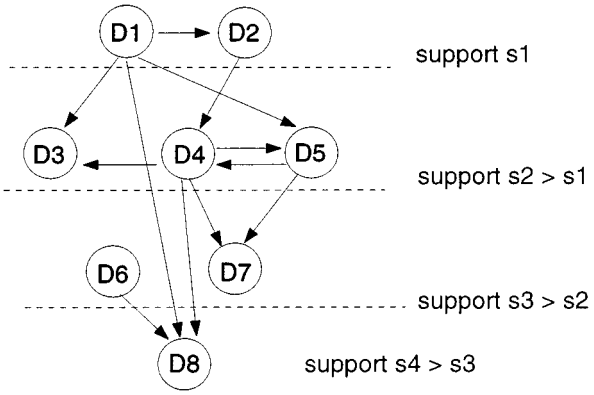

Figure 2. Typical graph for a disjunctive query. Nodes represent disjuncts, and edges represent the ReUse sets between source and destination.

Here is how we order the nodes for evaluation:

1. Disjuncts are evaluated level by level: all nodes with smaller support requirement are evaluated before nodes with the next higher support requirement.
2. Within each level, we do the following:

(A) if there are no cycles, we use the ReUse edges to do a topological sort, and evaluate the nodes from least to highest in-degree. At any step (say for in-degree k), if we have more than one choice, we rank them using the tie-breaker, and consider them ¿from highest to lowest value of tie-breaker.

(B) If there are cycles within a level, for each cycle, we use the tie breaker to rank the nodes within a cycle, then evaluate them from highest to lowest value of the tie-breaker.

The tie-breaker is a simple heuristic function, which computes the following expression for a node at level i:

$$(IN_{(i-1)} + OUT_i) - (IN_i + OUT_{(i+1)})$$

where $IN_{(i-1)}$ is the number of incoming reuse patterns from top level, OUT_i is the number of outgoing patterns within the same level, IN_i is the number of incoming patterns from within the same level, and $OUT_{(i+1)}$ is the number of outgoing patterns to the next level.

The intuition behind the tie-breaker is as follows: Nodes which are using maximum conjunct-patterns from the previous levels should be given priority, lest we are forced to swap them out while evaluating other nodes. By contrast, if a node generates a large number of conjunct-patterns to be used in the next level, it better be evaluated as close to the end of the current level, to maximize the chances of the conjunctsets being in memory when we get to the next level (since all nodes of the current level must be processed before then). In

addition, nodes which generate more conjunctsets for use by the same level should be done earlier, so the other nodes may benefit.

In the interest of simplicity, the above heuristic treats each pattern to be the same weight, when weighing a reuse edge. However, the same intuition can be developed into a cost-based optimization algorithm, much like used by relational databases today, if we considered the weight, and actual memory usage of each conjunct-pattern, and how much of it was being re-utilized by the consumer disjunct(s). Needless to say, doing that would increase the complexity of the query processing algorithm lot, and would preclude full compile-time ordering of disjuncts, since assigning real cost to the edges would require run-time examination of generated conjunctsets and a much closer syntactic examination of the query patterns. It is a reasonable problem, to be studied as part of future work.

13.3.1. Swapping priority A related issue to the one above, is that of deciding which conjunct-patterns to swap out of memory, when it becomes necessary to do so for the algorithm to make progress.

In this case, the algorithm follows a simple scheme. The priority to "stay in memory" is given to the conjunct-patterns likely to be used "most recently", with respect to evaluation order. This is similar to the optimal page replacement algorithm in operating systems research, which simply states that "pages which are not going to be used for the longest amount of time should be swapped out, if necessary." In operating systems, however, no page replacement scheme matches the performance of the optimal algorithm, since that requires some degree of clairvoyance. (The least recently used (LRU) algorithm, however, closely approximates the performance of the optimal algorithm).

In our particular case however, we can apriori know the order in which the nodes (disjuncts) will be evaluated by the algorithm. Therefore, when there is the need to swap conjunct-patterns out of memory, we select the ones which will be used by the nodes to be evaluated farthest in time.

13.4. The algorithm

Having understood the above issues of constraint distribution, conjunctset reuse and conjunctset replacement policies, the algorithm itself is easy to follow. The steps to the algorithm are described below:

1. Arrange the "where" clause in DNF (disjunctive normal form)[3]. In other words, the query looks like $(D_1 \vee D_2 \vee .. \vee D_n)$, where each disjunct can only be made up of conditions involving Body, Consequent, support, confidence, length and mutex logically "AND"-ed together. This step effectively distributes any pruning constraints over the disjunctions as discussed in section 13.1.
2. Consider the disjuncts pairwise. For any pair of disjuncts (D_i, D_j), evaluate *ReUse* (D_i, D_j) and *ReUse*(D_i, D_j). If either one is non-empty, draw an edge from the respective source to the respective target, and label it with the conjunct-patterns which can be recycled. Assign each edge with a weight equal to the number of conjunct-patterns in the respective *ReUse* set.

3. Perform a depth-first traversal on this forest of nodes. It may yield a set of more than one completely disconnected graphs. What this means is that the different graphs thus formed don't share any subexpressions and can be executed totally independently. This has the nice effect of reducing the memory requirements of the complete query.

   Denote each independent subgroup of nodes a subquery. For each such subquery, execute the next step.
4. Consider the nodes in the graph one at a time, using the evaluation order developed in Section 13.3. When faced with the decision to swap conjunct-patterns, use the optimal algorithm for swapping conjunct-patterns, as discussed in the last section.

The ideas developed for effectively evaluating multiple disjuncts of a query, notably the construction of ReUse sets, are not limited to "within" a query, but can also be deployed across independent queries posed by multiple users in a KDMS setting. In Virmani (1998), when we talk about "mining sessions" and batch-jobs submitted in different sessions for evaluation, the set of all queries submitted by different users can be treated as the set of disjunctions of a large query, and the ideas of conjunctset caching and reusing applied to them in a similar fashion.

14. Comparison with other rule query languages

We are aware of several other efforts in addition to ours, in terms of providing a declarative query format to generate and manipulate association rules.

DMQL, proposed by Han et al. (1996) allows declarative specification of several rule generating tasks, including generation of association rules, discriminant rules, classification rules, and characteristic rules. The constructs for the above tasks appear to "generate" knowledge, while allowing for expressions to pre-process the data. The language also contains commands to specify concept hierarchies at intensional or extensional level and apply them during mining tasks. However, we are not aware of similar operations to query, or post process the results of mining in the language. The language works on relational databases with discrete attributes. For continuous attributes, a hierarchy could be externally defined and specified in the mining query.

The other effort, presented by Meo et al. (1996) proposes the MINE RULE operator, which follows an SQL like syntax to generate rules. The work in Meo et al, (1996) reformulates several typical rule generation queries into the MINE RULE format, and provides procedural semantics of the operator. In contrast with the GETRULES operator presented in this paper, the MINE RULE operator works on the market basket data, kept in the normalized relational format with one row per item purchased per transaction. The MINE RULE operator is proposed as an extension to SQL, and is used to generate rules only, as opposed to querying persistently stored rules.

Sarawagi et al. in (1998) describe various levels of coupling mining systems with databases, one of which involves embedding computation into a DBMS engine by extending sql-92 with several primitives such as KWayJoin, and several other others such as Gather-Join, GatherPrune in the area of object-relational extensions to SQL. The work also draws comparisons to the approach of mining on data cached in a file system, and to the approach of

pushing user-defined functions (UDFs) into the mining algorithms implemented in SQL. Although not fully language related, has good bearing on the performance and scalability of various language extensions.

There has also been some effort in integrating deductive database languages with data mining. KnowledgeMiner is a data mining system, developed by Shen et al. (1996), which uses meta-rules based on LDL^{++} to guide the data mining process. More recently, in (Tsur et al., 1997) the authors have used Datalog to express query flocks: a generate-and-test model for data mining problems. Our method differs from those works as we have focused our language to be close to SQL, allowing easy adaptations for a user.

15. Conclusion

Much of the research in this area has focussed on "*massive rule generation*", i.e. generating all rules above a certain confidence and support. The algorithms have accordingly been performance-tuned for that task. Given the large number of rules produced in such an endeavor, we feel that a powerful, expressive language is essential to query the generated set of rules. In addition, the same language can be used to specify selective, query based generation of rules from data.

In this paper, we identified the key requirements of such a language, and presented the design, syntax, and compilation of MSQL and SQL based language used in the *Discovery Board* data mining system. We presented algorithms for evaluating basic MSQL queries and for more complicated cases involving arbitrary SQL nesting. We also identified several scenarios where interesting optimizations are possible due to the various types of nested query formats. We finally compared our approach with other approaches in literature.

At this point, a limited subset of MSQL has been implemented in the *Discovery Board* system. The full implementation of the above design will shortly be integrated into the system.

Notes

1. We also allow descriptor-patterns of the form (method=*) in the must set. This is interpreted as meaning that a rule may have any value for this method in the Body.
2. One must be careful to simply *add* and not *replace* the inner length-clause, since that will change the semantics of the inner query!
3. This may, in the worst case, yield an exponential increase in the number of terms in the query, but the increase is only in the size of the original query, which on an average is going to be fairly small (rarely more than ten conditions).

References

Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining associations rules between sets of items in large databases. Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'93), Washington D.C., pp. 207–216.

Han, J., Fu, Y., Koperski, K., Wang, W., and Zaiane, O. 1996. DMQL: A data mining query language for relational databases. DMKD-96 (SIGMOD-96 Workshop on KDD), Montreal, Canada.

Imielinski, T. and Mannila, H. 1996. A database perspective on knowledge discovery. Communications of the ACM, 39(11):58–64.

Meo, R., Psaila, G., and Ceri, S. 1996. A new sql-like operator for mining association rules. Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96), Bombay, India, pp. 122–133.

Ng, R.T., Lakshmanan, L.V.S., Han, J., and Pang, A. 1998. Exploratory mining and pruning optimizations of constrained associations rules. Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'98), Seattle, Washington.

Sarawagi, S., Thomas, S., and Agrawal, R. 1998. Integrating association rule mining with relational database systems: Alternatives and implications. Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'98), Seattle, Washington.

Shen, W.M. Ong, K., Mitbander, B., and Zaniolo, C. 1996. Metaqueries for data mining. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, (Eds), Advances in Knowledge Discovery and Data Mining, Menlo Park, CA: AAAI Press.

Tsur, D., Abbiteboul, S., Clifton, C., Motwani, R., and Nestrov, S. 1997. Query flocks: A generalization of association rules. Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'98), Seattle, Washington.

Virmani, A. 1998. Second generation data mining: Concepts and implementation. PhD Thesis, Rutgers University.

Tomasz Imielinski is currently Professor and Chairman of the Department of Computer Science, Rutgers University in New Brunswick NJ, USA. He has received his PhD from Polish Academy of Science (Warsaw) in 1982. His current interests include database mining and mobile wireless computing. He has published nearly 100 papers and coedited three books in these areas, including the joint paper with Rakesh Agrawal and Arun Swami (SIGMOD 1993) which first introduced association rules. Dr. Imielinski is on editorial boards of several journals and has been involved in organization and program commitees of several coneferences, including being the Program chair of the ACM Mobicom 1999 conference, in Seattle, Washington.

Aashu Virmani is a Member of Technical Staff at Bell Labs, NJ. He received his PhD in Computer Science from Rutgers University, NJ, in April '98. As part of his thesis work, he designed and developed the second generation data mining system "Discovery Board", that has been part of several publications and conference demonstrations. Prior to that, he worked at Bellcore where he developed "SCOUT"—a system to perform dissemination of real-time traffic information to motorists. His current focus involves developing policy based network management systems for IP telephony switching systems, where policies are dervied by applying past experience and by mining large amounts of historical call-logging data.