Decision Tables: Scalable Classification Exploring RDBMS Capabilities

Hongjun Lu
Department of Computer Science
Hong Kong University of Science & Technology
Hong Kong, China
luhj@cs.ust.hk

Hongyan Liu
School of Economics and Management
Tsinghua University
Beijing, China
liuhy@em.tsinghua.edu.cn

Abstract

In this paper, we report our success in building efficient scalable classifiers in the form of decision tables by exploring capabilities of modern relational database management systems. In addition to high classification accuracy, the unique features of the approach include its high training speed, linear scalability, and simplicity in implementation. More importantly, the major computation required in the approach can be implemented using standard functions provided by the modern relational DBMS. This not only makes implementation of the classifier extremely easy, further performance improvement is also expected when better processing strategies for those computations are developed and implemented in RDBMS. The novel classification approach based on grouping and counting and its implementation on top of RDBMS is described. The results of experiments conducted for performance evaluation and analysis are presented.

1. Introduction

Classification is a process of finding the common properties of data objects that belong to the same *class*. It has a wide range of applications, such as credit approval, customer group identification, medical diagnosis, etc. The problem has been studied extensively by researchers in various fields, such as statistics, machine learning, and neural networks [WK91]. During the recent surge of KDD research, classification becomes one of the most studied data mining problems [AIS93b]. Techniques developed earlier were re-examined in the new context [AGI+92, LSL95, MAR96, SAM96, WIV98, LHM98]. Since classification algorithms developed in machine learning and statistics assume that the training set resides in memory, most of the recent work is devoted to develop scalable classifier for training set whose size is much larger than the size of memory.

The work reported in this paper is motivated by the following observations.

First, most existing scalable classification algorithms [MAR96, SAM96, WIV98] are decision tree based [Quin93]. Decision tree based algorithms consist of two phases: *tree building* and *tree pruning*. During the tree-building phase, the training set is split into two or more partitions using an attribute (the *splitting attribute*). This process is repeated recursively until all (or most of) the examples in each partition belong to one class. Both the selection of the splitting attribute and the splitting points involve operations with high computational cost such as scanning the data, sorting and subset selection. Furthermore, since such operations are required at each internal node, they become the performance bottleneck of scalable classification [MAR96].

Second, although most work on scalable classification algorithms originated from the database researchers, the database technology developed during the past decades has not been fully explored in developing efficient scalable classifiers. Recently, researchers have started to focus on issues related to integrating data mining with databases. Agrawal *et. al.* addressed issues of tightly coupling a mining algorithm with a relational database system from the system point view [AS96]. They propose to use user-defined functions (UDFs) in SQL statement to push parts of the computation required by data mining

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

algorithms into the database system. Wang, Iyer and Vitter reported their experience in mining classification rules in relational databases using UDFs [WIV98]. Since UDFs are not native functions to DBMS, the performance benefit that the relational DBMS can provide may still be limited.

Finally, in order to meet the challenge posed by on-line analytical processing (OLAP) of large volume of data, efficient processing of data aggregation and summarization has been a research hit in recent years. Extensions to RDBMS functionality, operations and the related process strategies have been proposed [GCB+97, AAD+96, ZDN97]. The results have been quickly integrated into the commercial products. For example, IBM DB2 extended the traditional GROUP BY functions to include GROUP BY GROUPING SETS, GROUP BY CUBE, GROUP BY ROLLUP [Cham98]. Although such data aggregation and summarization are basic operations of a classification process, those new developments have not attracted much attentions from the data miners yet.

Based on the above observations, we developed a new approach, *GAC* (***G**rouping **A**nd **C**ounting*), for scalable classification. Like any classification process, the input of *GAC* is a training set, a set of objects with known classes, often in the form of (n+1)-tuples, $(a_1, a_2, ..., a_n, c_k)$, where a_i is a value from the domain of attribute A_i , $A_i \in \{A_1, A_2, ..., A_n\}$, and $c_k \in \{c_1, c_2, ..., c_m\}$ is the class label. The output of *GAC*, a *GAC*-classifier, is a table, *decision table* with n+3 columns, $(A_1, A_2, ..., A_n, Class, Sup, Conf)$. Each row in the table, $(a_{1i}, a_{2i}, ..., a_{ni}, c_i, sup_i, conf_i)$, represents a classification rule

if $(A_1 = a_{1i})$ and $(A_2 = a_{2i})$ and … and $(A_n = a_{ni})$

then $class = c_i$ $(sup_i, conf_i)$

where sup_i, and $conf_i$ give the *support* and *confidence* of the rule. The support of the rule denotes how popular is the rule and the confidence of the rule is the conditional probability, $P(class = c_i \mid (A_1 = a_{1i}) \wedge (A_2 = a_{2i}) \wedge \ldots \wedge (A_n = a_{ni}))$. The value of a_{ki} , $(1 \le k \le n)$, could be a "*don't care*" value, denoted by a special token *ANY* in our paper. In such case, term $(A_k = a_{ki})$ can be dropped from the rule.

The process of generating the decision table is rather simple and can be implemented using the powerful grouping and counting facilities provided by modern relational DBMS, such as IBM's DB2. After proper grouping and counting, the statistical information about class distribution over attribute values is obtained in a candidate decision table. The final decision table is obtained by pruning the entries in the candidate table.

The unique features of our approach include the following.

1. Compared to existing scaleable classifiers, the new approach not only provide as high classification accuracies as other approaches, such as naïve Bayesian classifier [DH73], Bayesian classifiers [FGG97], decision trees [Quin93], and classifiers based on associations [LHM98,MW99], but also improves the classification speed in the order of magnitudes. It also achieves the linear scalability with respect to the number of training samples within the tested range up to ten millions samples. We are able to obtain such drastic performance gain because we shifted away from the traditional record-at-a-time paradigm to the set-oriented relational paradigm. The main computation required, the grouping and counting over a large data set can be completed by executing a single SQL statement. With the current technique, such SQL statement can be executed rather efficiently even for large training set.

2. Our approach, GAC, can be implemented using standard data aggregation and summarization functions supported by RDBMS. As such the speed of building a decision table mainly depends on how the underlying DBMS process those operations. Whilst our experimental results indicate the current implementation such as in DB2 provides surprisingly good results, further performance improvement can be expected when better processing algorithms are developed and implemented in DBMS. Another benefit of avoiding UDFs is that the implementation becomes much easier. This can be seen clearly if we compare our implementation with MIND, a scalable miner for database implemented using user defined functions [WIV98].

3. Although our approach bears some similarities with those association rule based classification algorithms [AMS97, Bay97, LHM98, MW99], our approach avoids the Apriori-based frequent rule set finding [AIS93a], which requires to transform the training data into to transactional database and to scan the data repeatedly. The performance comparison with CBA [LHM98] indicates that our approach is more than 10 times faster.

The remainder of the paper is organized as follows. Section 2 describes the grouping and counting based classification approach in general. Section 3 presents the detailed implementation of the approach on top of relational database management systems. The results of our performance evaluation are presented in Section 4. A brief discussion on related work is presented in Section 5. Section 6 concludes the paper.

2. Decision Table and Its Generation

In this section, we introduce a new type of classifier, the *decision tables* and a grouping-and-counting based approach that generate decision tables for given data sets. The related issues will also be discussed.

2.1 An Illustrative Example

Before we formally describe the grouping and counting based classification approach, we use a mini-training data about car insurance shown in Table 2.1 to illustrate the basic idea. The table consists of three columns, two attributes, *age-group* and *car-type*, and a class label *risk*.

The objective of the classification problem is to find the rules that can be used to determine the class of a customer based on their age group and the types of cars they own.

Table 2.1: An example training set

age-group	car-type	risk
young	family	high
young	sport	high
middle	sport	high
old	family	low
middle	family	low

From the training samples in Table 2.1, we can obtain the class population for each combination of attribute values by grouping and counting as shown in Column 2 – 5 of Table 2.2. For easy reference, we added the row number as the first column. Column 6 and 7 are derived from Column 2- 5 and will be explained later. In this table, the grouping attributes of each row consist of one or more attributes and the class label. Row 1-4, 5-7 and 8-12 are grouped on (*age-group, risk*), (*car-type, risk*), and (*age-group, car-type, risk*), respectively. Column 5, *c-count* (class count), contains the total number of tuples whose attribute values are the same as the grouping attribute values. For example, *c-count* = 2 for Row 1 means that there are two tuples with *age-group* = *"young"* and *risk* = *"high"* in the training set; and *c-count* = 1 for Row 2 means that there are only one tuple with *age-group* = *"middle"* and *risk* = *"high"*, and so on.

Table 2.2: Results of grouping and counting

row No.	age-group	car-type	risk	c-count	g-count	conf
1	young		high	2	2	1.00
2	middle		high	1	2	0.50
3	middle		low	1	2	0.50
4	old		low	1	1	1.00
5		Family	high	1	3	0.33
6		Family	low	2	3	0.67
7		sport	high	2	2	1.00
8	young	Family	high	1	1	1.00
9	young	sport	high	1	1	1.00
10	middle	sport	high	1	1	1.00
11	middle	Family	low	1	1	1.00
12	old	sport	low	1	1	1.00

From Column 2-5, two additional columns, *g-count* (*group count*) and *conf* (*confidence*), can be computed: Column *g-count* contains the number of tuples in the same group, i.e., tuples with the same attribute values (exclusive of class label). For example, *g-count* = 2 in Row 1 means that there are two tuples with *age-group="young"* and *g-count=1* of Row 8 indicates that there is only one tuple with *age-group="young"* and *cat-type="family"*. Note that if the original table is sorted on grouping attributes, the value of *g-count* can be obtained after reading all the rows in the same group. The figures in the last column, *conf*, is obtained by dividing the two counts, *conf* = *c-count*/*g-cou*nt. That is, the value of *conf* represents the conditional probability for a tuple having the indicated class label given its attribute values. In our example, *conf* =1.00 for Row 1 means that, if the value of attribute *age-group* of a tuple is *young*, the probability that the tuple belongs to class *risk="high"* is 100%. In other words, rows in Table 2.2 can be interpreted as classification rules with certain level of *support* and *confidence*. For example, the first row in Table 2.2 represents the following rule:

age-group = "young" → *risk = "high"*

with 100% confidence and 40% (2/5) support from the training data in Table 2.1.

The confidence of certain row is quite low. For example, the confidence of second row is only 0.50. That is, given a tuple with age-group = "middle", we are not able to tell the class labels. There is another type of rows, such as Row 8. It represents the following rule

(*age-group="*young"), (*car-type = "family"*) → *risk = "high"*

with confidence equal to 1.00. However, this rule is redundant since if we already have the rule generated from Row 1.

Table 2.3: The decision table for the sample data

age-group	car-type	risk	sup	conf
young	***ANY***	high	0.40	1.00
ANY	sport	high	0.40	1.00
old	***ANY***	low	0.20	1.00
middle	family	low	0.20	1.00

If we delete all those rows in Table 2.2 with confidence less than 1.0 and rows that represent redundant rules, we obtained Table 2.3. The first three columns are the attributes and class label as in the training data. Column *sup* is the support of the rule represented by the row, obtained by dividing the *c-count* in Table 2.2 by the total number of tuples in the training data, 5 in our example. Last column is the confidence explained above. We name the table as a *decision table*, as each row in the table represents a rule that can be used to determine the class of a sample with given attribute values. In our example, the table contains the following rules.

age-group = "young" → *risk = "high"* (40%, 100%)
age-group = "old" → *risk = "low"* (40%, 100%)
car-type = "sport" → *risk = "high"* (20%, 100%)
(age-group ="middle"), (car-type = "family")
→ *risk = "low"* (20%, 100%)

2.2 Decision Tables

In this subsection, we define decision table and discuss how it can be used to classify unknown sample.

Definition. *Decision table* for data set *D* with *n* attributes A_1, A_2, ..., A_n is a table with schema *R* (A_1, A_2, ..., A_n , *Class, Sup, Conf*). A row R_i = (a_{1i}, a_{2i}, ..., a_{ni} , c_i, sup_i, $conf_i$) in table *R* represents a classification rule, where a_{ij} $(1 \le j \le n)$ can be either from *DOM*(A_i) or a special value *ANY*, $c_i \in \{ c_1, c_2, ..., c_m \}$, $minsup \le sup_i \le 1$, and $minconf \le conf_i \le 1$ and *minsup* and *minconf* are predetermined thresholds. The interpretation of the rule is **if** ($A_1 = a_1$) and ($A_2 = a_2$) and … and ($A_n = a_n$) **then** *class* $= c_i$ **with probability** $conf_i$ **and having support** sup_i , where $a_j \neq ANY$, $1 \le j \le n$.

Example: Table 2.3 is the decision table for data given in Table 2.1 with *minsup* = 0.20 and *minconf* = 1.00.

Since a row in a decision table represents a classification rule, we will use these two terms interchangeably in this paper.

Definition. A tuple $t = (a_1, a_2, ..., a_n , c_k)$ *matches* $R_i = (a_{1i}, a_{2i}, ..., a_{ni} , c_i, sup_i\ \mathrm{conf}_i)$ if for all a_j , $(1 \le j \le n)$, either $a_{ji} = ANY$ or $a_j = a_{ji}$. If tuple *t* matches R_i and $c_k = c_i$ we say that tuple *t is covered* by R_i, or R_i *covers t.*

Example: (*young, family, high*) matches and is covered by (*young, ANY,* high, *sup, conf*).

Given $R_i = (a_{1i}, a_{2i}, ..., a_{ni} , c_i, sup_i, conf_i)$ and $R_j = (a_{1j}, a_{2j}, ..., a_{nj} , c_j, sup_j, \mathrm{conf}_j)$, we say R_j is *redundant* if all the tuples covered by R_j are covered by R_i .

Lemma. Given two rules, $R_i = (a_{1i}, a_{2i}, ..., a_{ni} , c_i, sup_i, conf_i)$ and $R_j = (a_{1j}, a_{2j}, ..., a_{nj} , c_j, sup_j, conf_j)$, If (1) $c_i = c_j$, and (2) $a_{lj} = a_{li}$ for all $1 \le l \le n$ if $a_{li} \neq ANY$, then rule R_j is redundant.

Proof: Let $(a_{1k}, a_{2k}, ..., a_{nk}, c_k)$ be a tuple covered by R_j. Based on the definition, we have $a_{lk} = a_{lj}$ for all $1 \le l \le n$ if $a_{lj} \neq ANY$ and $c_k = c_j$.

(1) Since we are given $c_i = c_j$, we have $c_i = c_j = c_k$.

(2) Since $a_{lj} = a_{li}$ for all $1 \le l \le n$ if $a_{li} \neq ANY$, we have $a_{lk} = a_{lj} = a_{li}$ for all $1 \le l \le n$ if $a_{li} \neq ANY$.

Therefore, $(a_{1k}, a_{2k}, ..., a_{nk}, c_k)$ is covered by R_i . Since any tuple covered by R_j is covered by R_i, R_j is redundant.

Example: Given (*young, ANY, high*), (*young, family, high*) becomes redundant.

2.3 *GAC*: Generating Decision Table by Grouping and Counting

In this subsection, we describe *GAC*, a decision table generation algorithm based on grouping and counting. The input is a training data set, *D*, consisting of *N* tuples, each of which is an (*n*+1)-tuple, $(a_1, a_2, ..., a_n, c_k)$, where a_i is a value from the domain of attribute A_i , $DOM(A_i)$ and $A_i \in \{A_1, A_2, ..., A_n \}$, and $c_k \in \{ c_1, c_2, ..., c_m \}$ is the class label. We assume that all the attributes are categorical. Those non-categorical attributes are discretized using any appropriate existing discretization algorithms [FI93]. The output of *GAC* is a decision table. *GAC* consists of two major steps: *grouping and counting* and *table pruning*.

2.3.1 Grouping and counting

With a given training data set *D* (A_1, A_2, ..., A_n, *Class*), the grouping and counting phase generates a table that contains all possible entries in the decision table. We call this table a *candidate decision table*. In addition to the original columns in the data set, the candidate decision table has one more column, *count*, whose value is the number of tuples in the training data covered by the corresponding row in the table. That is, the candidate decision table has schema (A_1, A_2, ..., A_n, *Class, Count*).
The computation in this phase is rather straightforward. Tuples in the training data set are grouped based on their attribute values and class labels. For each grouping, the number of tuples that belong to each class is counted and recorded in the *count* column. For those non-grouping attributes, we use a special value *ANY* in the candidate decision table.

2.3.2 Table pruning

The size of the candidate decision table is usually large. Not all its rows should be used to form rows in the decision table. A row in the candidate table will be pruned if the classification rule it represents

1. is not statistically significant; or
2. has low confidence; or
3. is redundant.

The significance of a rule is measured by its support. With the given training data, the support of a rule, *sup*, is the number of tuples covered by the rule divided by the size of the data set. For a rule to be statistically significant, its support should be greater than a threshold, *minsup,* a parameter set by the system or user. Setting proper support threshold can prevent the problem of overfitting and increase the ability to handle noise data. If the threshold is set to less than 1/*N*, where *N* is the total number of tuples in the training data, then every tuple can be a classification rule even as it correctly classifies at least one sample. However, some tuple with very low support could be noise in the data.

Confidence of a rule represents the conditional probability of a tuple having the specified class label given its attribute values. By accepting a rule with confidence larger than a threshold, *minconf*, we actually allow that among a group of tuples with the same set of attribute values, (1-*minconf*) percentage of them have their class labels different from the class label for the group. That is, the threshold reflects our requirement of class purity. The appropriate value of *minconf* also depends on the application. For example, if training data contain noise, high *minconf* may result in no qualifying classification rules.

The third type of rows to be pruned from the candidate decision table is redundant rows, that is, the rows covered by the others in the table.

The main computational task for the second phase is to calculate the support and confidence for each row in the candidate decision table and remove entries that belong to the above three categories. Since the candidate decision table contains the counts for the number of tuples covered by a row in the table, the support is easy to calculate. Let's define a group as the tuples with the same non-*ANY* attribute values but different class labels. Then, we need only count the number of tuples of a group in order to calculate the confidence for the tuples in the group. Let the candidate decision table be sorted on (A_1, A_2, ..., A_n, *Class*) in ascending order; and the special value ANY is the minimum among all possible attribute values. It is obvious that a tuple can be only covered by tuples in the same group or the tuples in the previous groups in the sorted order. The decision table can be generated group by group as follows

(1) Tuples in the same group are read from the candidate decision table. The total number of tuples in the group is counted.
(2) The support and confidence for each tuple is calculated by definition.
(3) Tuples whose support or confidence is less than the specified threshold are discarded.
(4) Tuples with sufficient large support and confidence are checked against tuples already in the decision table. Redundant tuples are discarded. None redundant tuples are inserted into the decision table.
(5) Continue with the next group until all the tuples in the candidate decision table are processed.

2.3.3 An optimization

To reduce the size of the candidate decision table, we introduce one extra-step to determine the first attribute of the rules using methods that are used to determine the splitting attribute in decision tree based algorithms. Information gain is used as the goodness function of selecting the splitting attribute.

Assume attribute *A* has *k* distinct values. We will have *k* groups, D_1, D_2, …, D_k, if we group the training data tuples based on the values of *A*. The *information gain* for such a grouping is

$$I_A = E(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} E(D_i)$$

$$E(X) = -\sum_{i=1}^{m} \frac{count(c_i, X)}{|X|} \bullet \log \frac{count(c_i, X)}{|X|}$$

where $E(X)$ is the *entropy* of a set of tuples, *count*(ci, *X*) is the number of tuples in *X* that belong to class ci, $1 \le i \le m$, and $|X|$ is the total number of tuples in *X*. We choose among all the attributes the one that gives the largest information gain as the splitting attribute. In the first phase, we group and count tuples for groups with this attribute and others. That is, if there are four attributes, A, B, C, and D and C is the attribute giving the largest information gain among them, we only count for groups based on the values of (*C*, *A*), (*C*, *B*), (*C*, *D*), (*C*, *A*, *B*), (*C*, *A*, *D*), (*C*, *B*, *D*), and (*C*, *A*, *B*, *D*). Other groups, such as (*A*, *B*), (*A*, *B*, *D*), etc. will not be considered.

2.4 The Algorithm

Based on previous discussion, our algorithm, *GAC*, can be summarized as in Figure 2.1. The algorithm takes *TrainD*, the training data, and two thresholds, *minsup* and *minconf*, as input. The best splitting attribute is obtained by calling function *BestSplitAttr* that selects the best splitting attribute using information gain. Function *CandidateDTable* takes the training data and the splitting attribute as its input and generates the candidate decision table. The candidate decision table is pruned to form the decision table by calling function *PrunDTable* that takes *minsup* and *minconf* as input parameters.

```
Algorithm GAC (TrainD: table, minsup, minconf: real)
1   begin
2       bestSplitAttr := BestSplitAttr (TrainD);
3       candDTable := CandidateDTable (TrainD,
                          bestSplitAttr);
4       decisionTable := PrunDTable (candDTable,
                          minsup, minconf);
5   end.
```

Figure 2.1: The main algorithm that generates the decision table for a given data set.

2.5 Classification Using Decision Tables

The decision table generated is to be used to classify unseen data samples. To classify an unseen data sample, *u* (a_{1u}, a_{2u}, ..., a_{nu}), the decision table is searched to find rows that matches *u*. That is, to find rows whose attribute values are either *ANY* or equal to the corresponding attribute values of *u*. Unlike decision trees where the search will follow one path from the root to one leaf node, searching for the matches in a decision table could result in *none*, *one* or *more* matching rows.

One matching row is found: If there is only one row, r_i(a_{1i}, a_{2i}, ..., a_{ni} , c_i, sup_i, $conf_i$) in the decision table that matches *u* (a_{1u}, a_{2u}, ..., a_{nu}), then the class of *u* is c_i .

More than one matching row is found: When more than one matching rows found for a given sample, there are a number of alternatives to assign the class label. Assume that *k* matching rows are found and the class label, support and confidence for row *i* is c_i, sup_i and $conf_i$, respectively. The class of the sample, c_u, can be assigned in one of the following ways.

(1) based on confidence and support: $c_u = \{c_i \mid conf_i = \max_{j=1}^{k} conf_j\}$. If there are ties in confidence, the class with highest support will be assigned to c_u. If there are still ties, one randomly picked from them will be assigned to c_u.

(2) based on weighted confidence and support: $c_u = \{c_i \mid conf_i * sup_i = \max_{j=1}^{k}(conf_j * sup_j)\}$. The ties are treated similarly.

Note that, if the decision table is sorted on (*Conf, Sup*), it is easy to implement the first method. We can simply assign the class of the first matching row to the sample to be classified. In fact, our experiments indicated that this simple method provides no worse performance than others.

No matching row is found: In most classification applications, the training samples cannot cover the whole data space. The decision table generated by grouping and counting may not cover all possible data samples. For such samples, no matching row will be found in the decision table. To classify such samples, the simplest method is to use the default class. However, there are other alternatives. For example, we can first find a row that is the nearest neighbour (in certain distance metrics) of the sample in the decision table and then assign the same class label to the sample. The drawback of using nearest neighbour is its computational complexity.

Recall that we calculated class population for individual attributes in function *BestSplitAttr* to determine the best split attribute. With such information, we can use a Naïve-Bayesian based approach to determine the class as follows. Let u ($a_{1u}, a_{2u}, ..., a_{nu}$) be an uncovered sample, and $P(c_k \mid u)$ be the probability that u belongs to class $c_k \in \{c_1, c_2, ..., c_m\}$. According to Bayes theorem, we have

$$P(c_k|u) = \frac{p(c_k)p(u|c_k)}{P(u)}$$

That is,

$$P(c_k|u) \propto p(c_k)p(u|c_k)$$

With independence assumption we have

$$P(c_k|u) \propto p(c_k)\prod_{i=1}^{n} p(a_{iu}|c_k) = \frac{\prod_{i=1}^{n} p(a_{iu} \wedge c_k)}{p(c_k)^{n-1}}$$

since

$$p(a_{iu}|c_k) = \frac{p(a_{iu} \wedge c_k)}{P(c_k)}$$

For a given training data set, D, $P(a_{iu} \wedge c_j)$ and $P(c_j)$ can be approximate by the number of occurrences:

$$P(a_{iu} \wedge c_j) = \frac{count(a_{iu} \wedge c_j)}{|D|} \qquad P(c_j) = \frac{count(c_j)}{|D|}$$

Note that *coun t*($a_{iu} \wedge c_j$) and *count*(c_j) have been obtained in computing the information gain and determining the best splitting attribute. Therefore we can classify u into class c_k such that $P(c_k \mid u)$ is the maximum for $1 \le k \le m$. Experiments conducted indicated that classifying samples that do not have matches in the decision table using this approach provides higher classification accuracy than other ways listed.

3. Implementing GAC on Top of RDBMS

The approach described in Section 2 was motivated by the recent extensions of RDBMS capabilities. Traditionally, relational database management systems provide aggregate functions (MIN(), MAX(), AVG(), SUM(), COUNT()) and the GROUP BY operator to produce aggregates over a set of tuples. With applications of relational databases in on-line decision support systems, more complex aggregate functions and operators are required and proposed [GCB+97]. For example, recent releases of IBM DB2 provide a more powerful GROUP BY operator, which makes it possible to build a *GAC* classifier using SQL query language for the required major computation – the grouping and counting. In this section we describe in detail how SQL can be used to implement a classifier.

3.1 GROUP BY Operator In DB2

The traditional GROUP BY operation operates on a set of attributes. The semantics of the GROUP BY operator is to partition a relation (or sub-relation) into disjoint sets based on the values of the grouping attributes, the attributes specified in the GROUP BY clause. Aggregate functions are then applied to each of such sets. For example, SQL query

```
SELECT risk,  count(*)  FROM insurance
GROUP BY risk
```

partitions the relation insurance based on the values of *insurance.risk*, and counts the number of each partitions and produces a table with two columns, *risk* and the *count*. If the relation contains instances as shown in Table 2.1, the query will produce an output relation with two tuples (*high*, 3), (*low*, 2).

DB2 extended the GROUP BY operator to allow complex grouping requirements with *grouping-sets* and *super-groups*. A grouping-sets specification allows multiple grouping clauses to be specified in a single statement. By applying query

```
SELECT age-group, car-type, risk, count(*) as count
FROM insurance
GROUP BY GROUPING SETS   (age-group, car-type), risk
```

to our sample *insurance* table. We can obtain the following table:

age-group	car-type	risk	count
middle	-	high	1
middle	-	low	1
old	-	low	1
young	-	high	2
-	family	high	1
-	family	low	2
-	sport	high	2

For super-grouping, DB2 supports two super-groups, ROLLUP and CUBE. A CUBE grouping is the *n*-dimensional generalization of the GROUP BY operator. It can be viewed as a series of grouping-sets, i.e., all permutations of attributes in the GROUP BY CUBE clause are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2^n grouping-sets. For example, a GROUP BY CUBE (*age-group*, *car-type*, *risk*) query computes 8 grouping-sets: (*age-group*, *car-type*, *risk*), (*age-group*, *car-type*), (*age-group*, *risk*), (*car-type*, *risk*), (*age-group*), (*car-type*), (*risk*), and (), where GROUP BY () computes the aggregate function over the entire table. In addition to simple grouping-sets and super-groups, DB2 also supports the combination of such simple grouping. When simple grouping attributes are combined with other groups, they are "appended" to the beginning of the resulting grouping sets. When super-groups are combined, they operate like multipliers on the remaining groups, forming additional grouping set entries according to the definition of the super groups. For instance, GROUP BY *age-group*, CUBE (*car-type*, *risk*) will produce groups (*age-group*, *car-type*, *risk*), (*age-group*, *car-type*), (*age-group*, *risk*), and (*age-group*).

3.2 Building Decision Tables Using GROUP BY Operators

With the above introduction, we can see that the grouping and counting over the training data can be fully implemented using GROUP BY operator provided by DB2. However, GROUP BY CUBE is an expensive operation, especially when the number of attributes of the cube is large. More importantly, resource requirement for the computation is high. On the other hand, most classification rules do not involve all attributes A_1, A_2, …, and A_n. In order to improve efficiency, the basic algorithm shown in GAC is modified in such a way that the decision table is constructed in iterations. In each iteration, a number of attributes are used to generate entries in the decision table. To control the number of attributes used in cube computation, two system parameters, *initCubeSize* and *maxCubeSize*, are introduced to denote the number of attributes used in the first iteration and the maximum number of attributes to form cubes, respectively. The first parameter provides a mechanism to adjust the training speed. Intuitively, with large *initCubeSize*, less number of iterations will be required for training but each iteration takes longer time since large cube is to be computed. The second parameter is mainly determined by the system resource. The training process ends when the desired classification accuracy is achieved, or the predetermined training time limit is reached.

The algorithm is outlined in Figure 3.1. We assume that the discretized training data set is stored in a relational database as a table with schema (A_1, A_2, …, A_n, *class*). Two input parameters, *minsup* and *mincof* are given based on the application and the quality of the data. Another parameter minerror is used to control the training process. First, the attributes are sorted based on the information gain (line 2). The first attribute in the list, the one with highest information gain is chosen as the splitting attributes. Function *SelectAttrs* selects a set of attributes to generate decision table entries in each iteration (line 4). The decision table entries are generated as described in the previous section (line 5-6). The up-to-date decision table is applied to the training data set to check the error rate (line 7). The process repeats until the error is smaller than the required minimum error, or training time limit is reached (line 8). In the following subsections, we explain the details of the major functions.

```
Algorithm GAC-RDB (TrainD: table, minsup, minconf,
  minerror: real)

1. begin
2.   sortedAttrList:= SortAttr (TrainD);
3.   repeat
4.     curAttrs:= SelectAttrs (sortedAttrList);
5.     candDTable := CandidateDTable (TrainD, curAttrs);
6.     decisionTable := PrunDTable (candDTable, minsup,
                        minconf);
7.     error := EstimateError(TrainD, decisionTable);
8.   until error ≤ minerror or timeout
9. end.
```

Figure 3.1: *GAC-RDB*: A decision table generation algorithm.

3.2.1 Determining the first splitting attribute

As in algorithm *GAC*, we select an attribute as the splitting attribute based on information gain to reduce one attribute in the computation of cube, which is an expensive operation. We also use entropy to facilitate attribute selection in each iteration. Therefore, we sort the attributes on their entropy in descending order. The first attribute on the list is chosen as the split attribute explained in the previous section. The algorithm is outlined in Figure 3.2. The major computation is to calculate the class population among different attribute values. The SQL statement (line 3-6) performs this task. After obtaining these counts, entropy for partitioned data using each of the attributes is compared (line 7-8). The

```
Function SortAttr (TrainD: relation);
1.  begin
2.      obtain the class population for each attribute values
        by executing SQL query
3.      SELECT   $A_1, A_2, \ldots, A_n$, class, count(*)
4.      FROM     TrainD
5.      GROUP BY GROUPING STES
                 ( ($A_1$, class), ($A_2$, class), …, ($A_n$, class))
6.      ORDER BY $A_1, A_2, \ldots, A_n$
7.      for each attribute $A_i \in \{ A_1, A_2, \ldots, A_n\}$ do
8.          Compute entropy E($A_i$);
9.          Sort attributes on E($A_i$) into sortedAttrList ;
10.     return sortedAttrList;
11. end.
```

Figure 3.2: Function *SortAttr* that sorts attributes in the order of entropy.

attributes are then sorted in the descending order of their entropy.

As discussed in Section 2, the query results, i.e., the class population of each attribute value is saved to classify samples uncovered by decision table entries.

3.2.2 Selecting attributes

Given *initCubeSize* and *maxCubeSize*, Function *SelectAttrs* (line 4 in Figure 3.2) selects attribute set, *curAttrs*, for generating decision table entries as follows:

1. For the first iteration, top *initCubeSize* attributes in the *sortedAttrList* are selected as *curAttrs*.
2. After each iteration, attributes that contribute to decision table entries are inserted into a set, denoted as *relevantAttrs*, whose size is denoted as *R*
3. When R is smaller than *initCubeSize*, *RelevantAttrs* and top (*initCubeSize* - *R*) attributes in the *sortedAttrList* form the set of *curAttrs*. If R is larger than *initCubeSize*, *curAttrs* will be formed by the *relevantAttrs* and next attribute in *the sortedAttrList.*
4. After R reaches *maxCubeSize*, *curAttrs* will be formed by the top *maxCubeSize* of attributes in *relevantAttrs* and next attribute in the *sortedAttrList*.

That is, the current set of attributes used in each iteration is selected in the greedy fashion based on the entropy of attributes. The number of attributes selected is limited to (*maxCubeSize*+2).

This process is indeed a type of feature selection process [LM98]. However, it selects relevant features for a subspace defined by *curAtts* in each iteration, while most feature selection algorithms select relevant features based on entire training data. In most cases, not all classification rules involve all relevant features. Therefore, we can generate the whole decision table iteratively using different set of attributes.

3.2.3 Generating candidate decision table

Function CandidateDTable (line 5 in Figure 3.2) generates the candidate decision table with schema of (A_1, A_2, …, A_n, *class*, *count*) by executing the following SQL query:

```
SELECT    $A_1, A_2, \ldots, A_n$, class, count(*)
FROM      TrainD
GROUP BY  $A_k$, CUBE ($A_1, A_2, \ldots, A_{k-1}, A_{k+1} \ldots,$
              $A_n$), class
ORDER BY  $A_1, A_2, \ldots, A_n$
```

where A_k is the splitting attributes. Since the results obtained from the SQL query in Figure 3.2, line 3-6, may also be candidate classification rules, they are inserted into the candidate decision table. Since the candidate decision table will be processed by *PrunDTable,* it need not be stored on the disk. It can stay in the SQL common area.

3.2.4 Pruning the decision table

Implementation of Function *PrunDTable* is quite straightforward as described in the previous section. The decision table has one more field, *conf*, than the candidate decision table, with values equal the count of the rule divided by the sum of all counts of the rules from the same group. Therefore, the tuples in the candidate decision table are processed group by group. If a new tuple read is covered by the current group, it is inserted into the current group for processing and the count for the group is increased. When a tuple starts a new group, the old group is processed, i.e., tThe confidence of each rule in the group is computed. If the support and confidence of the rule are above the thresholds, and the rule is not covered by any other rules, the rule is inserted to the final decision table with its confidence.

4. A Performance Study

A comprehensive performance study has been conducted to evaluate the approach and our implementation. In this section, we describe those experiments and their results. All experiments reported in this section were performed on an HP Omnibook 3000 notebook computer with 233MHZ CPU running Microsoft Windows NT 4.0 and IBM DB2 version 5.0.

4.1 Classification Accuracy

Classification accuracy is one of the basic performance metrics for any classification algorithms. In their recent paper Meretakis and Wüthrich compared the classification accuracy of a set of classification methods [MW99] using a set of data from the UCI Repository [MM96]. We tested our system using the same data sets. The continuous attributes are discretized using the *MLC* discretizer based on entropy discretization [FI93, KJL+94]. To be comparable with the other methods, the size of training set and testing set follow what given in [MW99]. In most cases, 10-fold cross-validation is used. The results of our system are shown in Table 4.1, along with other five other classifiers with different approaches:

Table 4.1: Classification accuracy of GAC and other classifiers as given in [MW99]

Data set	**Properties**				**Accuracy**					
	#attrs	**#classes**	**# train**	**# test**	**C4.5**	**NB**	**TAN**	**CBA**	**LB**	**GAC**
Australian	14	2	690	CV-10	0.843	0.857	0.852	0.855	0.857	**0.883**
Chess	36	2	2,130	1,065	**0.995**	0.872	0.921	0.981	0.902	0.944
Diabetes	8	2	768	CV-10	0.717	0.751	0.765	0.729	0.767	**0.767**
Flare	10	2	1,066	CV-10	0.812	0.795	0.826	0.831	0.815	**0.843**
German	20	2	1,000	CV-10	0.717	0.741	0.727	0.732	0.748	**0.768**
Heart	13	2	270	CV-10	0.767	0.822	0.833	0.819	0.822	**0.838**
Letter	16	26	15,000	500	0.777	0.749	**0.857**	0.518	0.764	0.800
Lymph	18	4	148	CV-10	0.784	0.819	0.838	0.773	**0.846**	0.839
Pima	8	2	768	CV-10	0.711	0.759	0.758	0.730	0.758	**0.780**
Satimage	36	6	4,435	2,000	0.852	0.818	**0.872**	0.849	0.839	0.847
Segment	19	7	1,540	770	**0.958**	0.918	0.935	0.935	0.942	0.943
Splice	60	3	2,126	1,064	0.933	0.946	0.946	0.700	0.946	**0.956**
Shuttle-small	9	7	38,661	934	0.995	0.987	0.996	0.995	0.994	**0.998**
Vehicle	18	4	846	CV-10	0.698	0.611	**0.709**	0.688	0.688	0.681
Voting Records	16	2	435	CV-10	**0.957**	0.903	0.933	0.935	0.947	0.956
Waveform-21	21	3	300	4,700	0.704	0.785	0.791	0.753	**0.794**	0.761
Yeast	8	10	1,484	CV-10	0.557	0.581	0.572	0.551	**0.582**	0.574
AVERAGE					0.810	0.807	0.831	0.787	0.824	**0.834**

(1) *C4.5*, Quinlan's decision tree classifier C4.5 [Quin93], (2) NB: a Naïve Bayes classifier [DH73], (3) TAN, a state of the art Bayesian network classifier that relaxes the independence assumptions of Naïve Bayes by taking into account dependencies among pairs of non-class attributes [FGG97], (4) CBA, a classifier based on association rules [LHM98], and (5) LB, a naïve Bayes classifier using long items [MW99].

From Table 4.1, we can summarize the relative performance of GAC as follows:

Ranking	1	2	3	4	5	6
# Data Sets	8	5	1	2	1	0

that is, comparing with other five classifiers, *GAC-RDB* performs the best for 8 data sets and the second best for 5 data sets. It never performs worst in other cases. Astute readers may argue that the absolute accuracy rates listed in Table 4.1 may not precisely reflect the performance since they were not obtained using the same training and testing data sets. However, from those figures, we are confident to claim that *GAC-RDB* can provide at least the same level of accuracy as other popular classifiers.

4.2 Execution Speed And Scalability with Respect to Number of Training Samples

The second set of experiments investigates the execution speed and the scalability with respect to the number training samples. This set of experiments used the synthetic data and classification functions defined in [AIS93b]. Each record in the data set consists of 9 attributes, including *salary, commission, age, elevel, car, zipcode, hvalue, hyears* and *loan*.. There are 10 classification functions defined on these 9 attributes. To compare with the results reported in the literature, we present the execution speed on two functions, Function 5 and 10.

Function 5

Class A: $((age<40) \wedge (((50k \le salary \le 100k))\ ?$
$(100k \le loan \le 300k) : (200k \le loan \le 400k)))) \vee$
$((40 \le age<60) \wedge (((75k \le salary \le 125k))\ ?$
$(200k \le loan \le 400k) : (300k \le loan \le 500k)))) \vee$
$((age \ge 60) \wedge (((25k \le salary \le 75k))\ ?$
$(300k \le loan \le 500k) : (100k \le loan \le 300k))))$

Function 10:

$hyears < 20 \Rightarrow equity=0$
$hyears \ge 20 \Rightarrow equity=0.1 \times hvalue \times (hyeares-20)$
$disposable = (0.67 \times (salary + commission) -$
$5000 \times elevel + 0.2 \times equity - 10k)$
Class A: $disposable > 0$

Since *GAC-RDB* works with categorical attributes, the non-categorical attributes are discretized first. We used a simple equi-width method for discretization. As mentioned earlier, feature selection algorithms are usually used to select the relevant features before classification. Since Function 10 has 5 relevant attributes (salary, commission, ed_level, hyear, hvalue), we use data sets consisting of these 5 attributes for both tests. The number of training samples was varied from 0.5 to 10 million. The elapsed time measured is shown in Figure 4.1.

From the figure, we can see that linear scalability is achieved with respect to the number of training sample

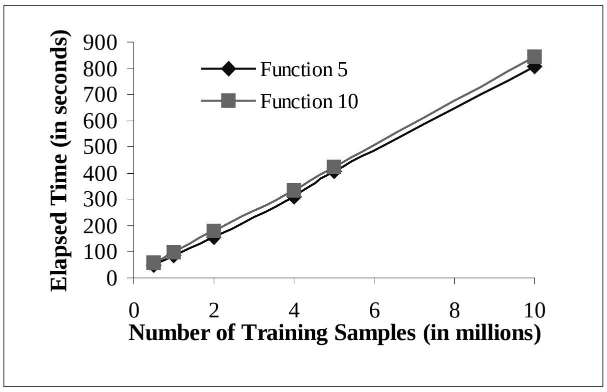


Figure 4.1: Elapsed time for Function 5 and 10

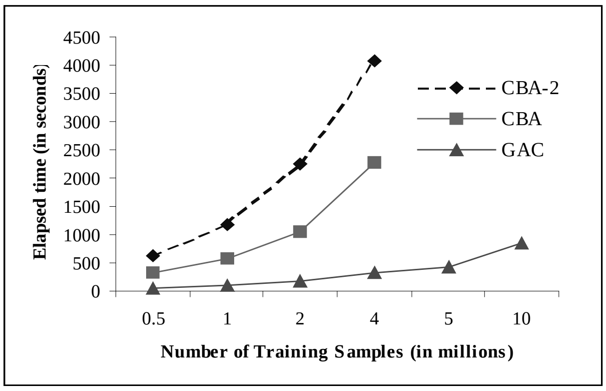


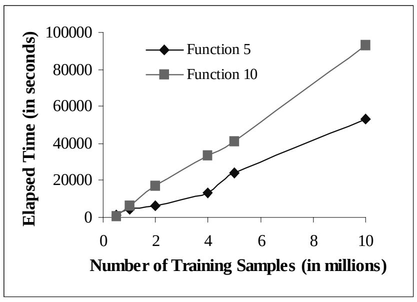
Figure 4.2: Comparison with CBA

within the range tested. To see how the execution speed compare to other algorithms, we obtained a copy of the executable code of *CBA* [LHM98] and run Function 10. By checking the log of *CBA* execution, we realized that the time reported by the program, indicated by curve *CBA* in Figure 4.2, is only the sum of scanning the data set. Furthermore, it does not include the time of transforming the data set into transactional database, which takes more than 300 seconds for 0.5 million tuples. It is reasonable to assume that the transformation is linear to the number of samples in the data set, the total elapsed time will then be at least as what shown by the curve *CBA*-2 in Figure 4.2. We can see that *GAC-RDB* is actually more than 10 times faster than *CBA*. Moreover, *CBA* failed to complete its execution for the data set of 5 and 10 million tuples in the notebook we conducted the tests.

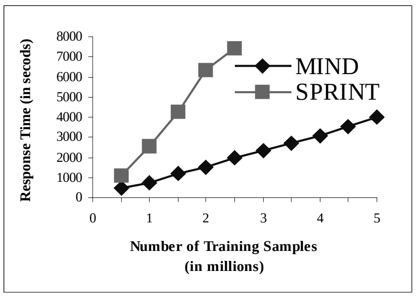
To compare with other scalable classifiers reported in the literature, we reproduced the two performance figures appeared in papers in Figure 4.3. Figure 4.3 (a) depicts the performance of SLIQ with two functions, Function 5 and 10 [Mar96]. Figure 4.3 (b) is scalability results of MIND for Function 2 reproduced according to Wang *et. al.* [WIV98]. While our results on Function 5 and 10 are given in Figure 4.1, the results on Function 2 are presented in Figure 4.3 (c).

Since performance figures for SLIQ, MIND and SPRINT were obtained running on IBM RS/6000 workstations running AIX, it is difficult to compare the absolute numbers. We would like to point out one important property of *GAC-RDB*. The classification speed of *GAC-RDB* is independent on the complexity of classification functions, as the major operation of *GAC-RDB*, grouping and counting only depends on the number of attributes, number of distinct values for each attribute and number of tuples. Comparing Function 5 and 10, we can see that Function 10 is non-linear which is more complex than Function 5 where the hyperplanes separating the classes are parallel to axis. With complex classification functions, the decision-tree will usually have more levels. Since the execution time of decision-tree based algorithms is directly related to the number of levels of the tree, classification of those functions requires longer time. This is clearly shown in the SLIQ performance for Function 5 and 10. On the other hand, performance of *GAC-RDB* is not affected by the classification functions.

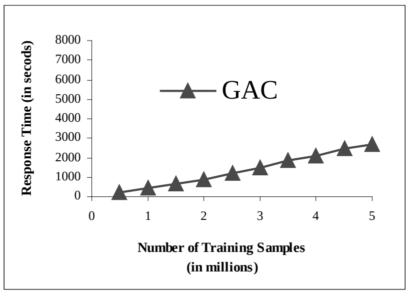
Function 2 reported in MIND is the simplest function among 10 classification functions: A sample belongs to class *if* ((age<40) ^ (50k≤$salary$ ≤100k)) ∨ ((40≤age<60) ^ (75k≤$salary$≤125k)) ∨ ((age≥60) ^ (25k≤$salary$ ≤75k)). Again, we can see that *GAC-RDB* performs well, comparing to other two algorithms.



(a) Results from SLIQ [MAR96]

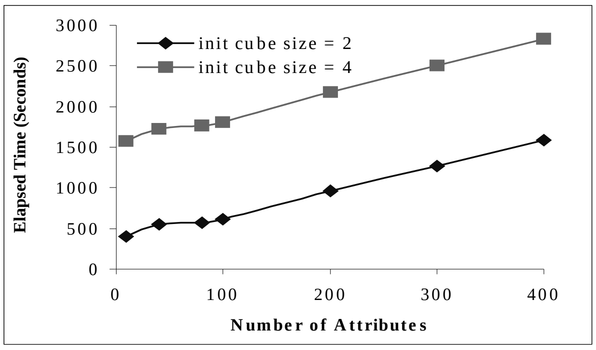


(b) Results reported in [WIV98]

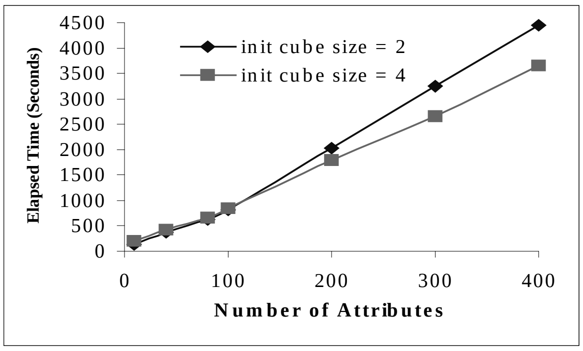


(c) Performance of GAC-RDB

Figure 4.3: Comparison with other scalable classifiers.



(a) Function 10



(b) Function 5

Figure 4.4: Scalability with respect to number of attributes.

4.3 Scalability With Respect to Number of Attributes

The third set of experiments was conducted to investigate the scalability of GAC-RDB with respect to the number of attributes. Function 5 and 10 were used in the experiments. The training data and testing data contain 100,000 and 10,000 samples, respectively. The number of attributes was varied from 9 to 400 by introducing extra attributes into the data set. Although the classification functions do not change when the number of attributes increases, the tuple length of the data increases. More importantly, with large number of attributes, it requires more time to locate the relevant attributes. Therefore, the training time is expected to increase. Figure 4.4 (a) and (b) depicts the elapsed time when the number of attributes increases from 9 to 400 for Function 10 and 5, respectively. For each case, we used two initial cube sizes, 2 and 4. The maximum cube size was set to 6.

We can see that, when the number of attributes increases, the elapsed time also increases. Comparing two data sets, the elapsed time for Function 5 increases much faster than that for Function 10 when the number of attributes increases from 9 to 400. However, for both functions, *GAC-RDB* provides near linear scale-up with respect to the number of attributes of training data.

In Figure 4.4, we also present the effects of the initial cube size on the training speed when the number of attributes increases. It is interesting to notice the different behavior of two functions. For Function 10, two curves for the initial cube size of 2 and 4 are in parallel. With large initial cube size, the cube to be computed at the beginning is large and it requires more computation time. When a data set contains large number of attributes, another factor will affect the speed of training. That is, the iterations need to include all relevant attributes into the decision tables. This can be seen from Figure 4.4 (b). With small number of attributes, setting small initial cube size has some advantage. However, when the number of attributes becomes large, larger initial cube size (4) leads to better performance. This also gives us some heuristics in setting the parameter of initial cube sizes.

One last note is that, all the experiments reported in this section were conducted on a notebook computer, which is not a really appropriate environment for large-scale computation tasks. For example, the resources are too limited to compile SQL queries with CUBE computation on large number of attributes. With more resources, *GAC-RDB* is expected to perform better.

5. Conclusions

Classification is a classical problem. It has been well studied by researchers from different areas. The book by Weiss and Kulikowski [WK91] gives the most comprehensive summary of the work in 1980s or earlier. A paper by Agrawal *et. al.* [AGI+92] trigged another round of interests in the classification problem, especially in the context of knowledge discovery and data mining. Since then, a large amount of work has been reported on building scalable classifiers and integrating classification into database systems to address the scalability problem. Recently, classification algorithms based on association rule mining were also introduced.

In this paper we described a novel approach to build efficient scalable classifiers by exploring the capability of relational database management systems that support powerful data aggregation and summarization functions. The approach is an elegant integration of all the recently developed techniques. As the result, it is scalable with respect to both the number of training samples and the number of attributes. Furthermore, with more sophisticate and efficient implementation of such data aggregate and summarization functions in relational DBMS, even better performance and scalability can be expected.

We are currently further refining the approach and carrying on more comprehensive performance evaluation. The issues related to feature selection, missing value and noise handling will also be addressed.

Acknowledgement

This work is partially supported by a grant from the National 973 project of China (No. G1998030414) and a grant from the Research Grant Council of the Hong Kong Special Administrative Region, China (AOE97/98.EG05)

References

[AAD+96] S. Agrawal, et. al., On the computation of multidimensional aggregates, In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996, 506-521.

[AGI+92] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of the 1992 International Conference on Very Large Databases*, Vancouver, Canada, August 1992, 560--573.

[AIS93a] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference of Management of Data*, Washington D.C., May 1993, 207-216.

[AIS93b] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6), December 1993.

[AMS97] K. Ali, S. Manganaris, and R. Srikant, Partial classification using association rules. In *Proceedings of 3rd International Conference on Knowledge Discovery and Data Mining*, 1997.

[AS96] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[Bay97] R.J. Bayardo, Brute-Force mining of high confidence classification rules. In *Proceedings of 3rd International Conference on Knowledge Discovery and Data Mining*, 1997.

[CFB99] S. Chaudhuri, U.M. Fayyad, and J. Bernhardt. Scalable classification over SQL databases. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 23-26 1999, 470-479.

[Cham98] D.D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann Publishers, 1998.

[DH73] R.Duda and P.Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, 1973.

[FGG97] N. Friedman, D.Geiger, and M. Goldszmidt, Bayesian network classifier, *Machine Learning*, 29, 1997, 131-163.

[FI93] U.M. Fayyad and K.B. Irani, Multi-Interval discretization of continuous-valued attributes for classification learning, In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993, 1022-1027.

[GCB+97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Richart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A relational aggregate operator generalizing group-by, cross-tab, and sub-totals, *Data Mining and Knowledge Discovery*, 1(1), 1997, 29-54.

[KJL+94] R. Kohavi, G. John, R. Long, D. Manley, and K. Pfleger, MLC++: a machine learning library in C++, *Tools with Artificial Intelligence*, 1994, 740-743.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, New York, USA, 1998, 80-86.

[LM98] H. Liu and H. Motoda, Feature extraction, construction and selection: A data mining perspective. Kluwer Academic Publisher, 1998.

[LSL95] H. Lu, R. Setiono, and H. Liu. NeuroRule: A connectionist approach to data mining. In Proceedings of *the 21th International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15 1995, 478-489.

[LSL96] H. Lu, S.Y. Sung, and Y. Lu. On pre-processing data for effective classification, *ACM SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, Canada, June 1996.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In Proceedings of the 5th International Conference on Extending Database Technology, Avignon, France, March 1996.

[MM96] C.J. Merz and P. Murphy, *UCI repository of machine learning databases*, 1996. (http://www.cs.uci.edu/~mlearn/MLRepository.html)

[MW99] D. Meretakis and B. Wüthrich. Extending naïve Bayes classifiers using long itemsets. In *Proceedings of 5th International Conference on Knowledge Discovery and Data Mining*, San Diego, California, August 1999.

[Quin93] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[SAM96] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996.

[WIV98] M. Wang, B. Iyer, and J.S. Vitter. Scalable mining for classification rules in relational databases. In *Proceedings of the 1998 International Database Engineering and Applications Symposium*, Cardiff, Wales, U.K., July 8-10, 1998.

[WK91] S.M. Weiss and C.A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufman, 1991.

[ZDN97] Y. Zhao, P.M. Deshpande and J.F. Naughton. An array-based algorithm for simultaneous multidimensional aggregations. In *Proceedings of the 1997 ACM-SIGMOD International Conference on Management of Data*, Tucson, Arizona, June 1997, 159—170.