Paper ID: 151

SQL Database Primitives for Decision Tree Classifiers

Kai-Uwe Sattler Oliver Dunemann
Department of Computer Science
University of Magdeburg
P.O.Box 4120, 39016 Magdeburg, Germany
{kus|dunemann}@iti.cs.uni-magdeburg.de

Abstract

Scalable data mining in large databases is one of today's challenges to database technologies. Thus, substantial effort is dedicated to a tight coupling of database and data mining systems leading to database primitives supporting data mining tasks. In order to support a wide range of tasks and to be of general usage these primitives should be rather building blocks than implementations of specific algorithms. In this paper, we describe primitives for building and applying decision tree classifiers. Based on the analysis of available algorithms and previous work in this area we have identified operations which are useful for a number of classification algorithms. We discuss the implementation of these primitives on top of a commercial DBMS and present experimental results demonstrating the performance benefit.

1 Introduction

The integration of data mining with database systems is an emergent trend in database research and development. A tight coupling between data mining and database systems is motivated by several observations. Today, most data mining systems process data in main memory. Though this results in high performance as long as enough memory is available, it ignores the fact that most data subject of analysis has been already stored in database systems and that database systems provide powerful mechanisms for accessing, filtering and indexing data. In addition, the main-memory or non-coupling approach suffers from the drawback of limited scalability. If the data set does not fit into the available memory the performance decreases dramatically. This problem has been addressed previously by data reduction techniques like sampling, discretization and dimension reduction. In contrast, SQL-aware data mining techniques could utilize sophisticated features available in modern DBMS like management of GB data sets, parallelization, filtering and aggregation and in this way improve the scalability. Another reason for building SQL-aware data mining systems is ad-hoc mining [Cha98], i.e., allowing to mine arbitrary query results and not only base data. So it would not be necessary to preprocess data just for applying mining operations. Instead the data set is created "on the fly".

However, a major drawback of SQL-aware data mining in today's DBMS is often poor performance. This is mainly due to the facts, that the rather simple SQL operations like join, grouping and aggregation are not sufficient for data mining. Therefore, data mining operations have to be implemented as series of SQL queries, which are treated by the DBMS normally isolated and independent from each other. In order to achieve a more efficient implementation, functionality of the data mining system should be pushed into the DBMS, allowing to utilize knowledge about the operations and their access patterns and paths. But because there are various data mining algorithms for different problems it is difficult to decide, which functionality should be integrated into the DBMS.

Based on this observation the main challenge is at first to identify data mining primitives and secondly to implement them using DBMS extension facilities, e.g. cartridges, extenders or datablades.

We can distinguish at least three levels of data mining support in DBMS:

(1) A first idea is adding new language constructs to SQL as proposed in [MPC96] for association rules.

(2) A second approach is to exploit data mining functionality implemented internally using a special API like OLE DB for Data Mining [NCBF00] or user-defined types and methods as proposed for SQL/MM Part 6.

(3) Finally, a DBMS could provide special operators or primitives, which are generally useful for data mining but not implementing a particular data mining task, e.g. the AVC sets described in [GRG98].

The advantage of approach (3) is the usefulness for a broader range of data mining functions and obviously, both the language and the API approaches could benefit from such primitives. Moreover, if we consider the complexity of the SQL-99 standard and the extent of the features currently implemented in commercial systems it should become clear, that the implementation of primitives based on available DBMS extension technologies seems to be the most promising approach.

In this paper we present results of our work on database primitives for decision tree classifiers. Classification is an important problem in data mining and well studied in the literature. Furthermore, there are proposals for classifier operations, e.g. [GRG98], which form the basis for our work. We extend the idea of computing AVC groups or CC tables respectively to implement a SQL operator for a commercial DBMS. We evaluate the benefit of multi-dimensional hashing for speeding up partial-match queries, which are typical queries in decision tree construction. Finally, we discuss the implementation of prediction joins – operators for applying an induced classification model on new data.

The remainder of this paper is organized as follows: In section 2 we recall the problem of decision tree classification, describe previous work and identify potential functions for database primitives. Section 3 describes these primitives in detail and discusses their implementation using the Oracle DBMS. In section 4 we report results of the evaluation of this implementation. Finally, we present related work in section 5 and conclude in section 6.

2 Decision Tree Classification

Classification is an important data mining problem that has been studied extensively over the years. So, several classification models have been proposed, e.g. bayesian classification, neural networks, regression and decision trees. Decision tree classification is probably the most popular model, because it is simple and easy to understand.

A number of algorithms for constructing decision trees are available including ID3, C4.5, SPRINT, SLIQ, and PUBLIC. Most decision tree algorithms follow a greedy approach, that can be described as follows [BFOS84]. In the *tree-growing* phase the algorithm starts with the whole data set at the root node. The data set is partitioned according to a splitting criterion into subsets. This procedure is repeated recursively for each subset until each subset contains only members belonging to the same class or is sufficiently small. In the second phase – the *tree pruning* phase – the full grown tree is cut back to prevent over-fitting and to improve the accuracy of the tree. An important approach to pruning is based on the minimum description length (MDL) principle [MRA95].

During the tree-growing phase the splitting criterion is determined by choosing the attribute that will best separate the remaining samples of the nodes partition into individual classes. This attribute becomes the decision attribute at the node. Using this attribute A a splitting criterion for partitioning the data is defined, which is either of the form $A < v, v \in dom(A)$) for numeric attributes or $A \in V$ ($V \subseteq dom(A)$) for categorical attributes. For selecting the best split point several measures were proposed, e.g. ID3 and C4.5 select the split that minimizes the information entropy of the partitions, while SLIQ and SPRINT use the gini index. For a data set S containing n records the information entropy $E(S)$ is defined as $E(S) = -\sum p_i \log_2 p_i$ where p_i is the relative frequency of class i. For a split dividing S into the subsets S_1 and S_2 the entropy is $E(S_1, S_2) = \frac{n_1}{n}E(S_1) + \frac{n_2}{n}E(S_2)$. The gini index for a data set S is defined as $gini(S) = 1 - \sum p_i^2$ and for a split $gini_{split}(S) = \frac{n_1}{n}gini(S_1) + \frac{n_2}{n}gini(S_2)$. Once an attribute is associated with a node, it needs not be considered in the node's children.

```
procedure BuildTree (data set $S$)
    if all records in $S$ belong to the same class
        return
    foreach attribute $A_i$
        evaluate splits on attribute $A_i$
    use best split found to partition $S$ into $S_1$ and $S_2$
    BuildTree ($S_1$)
    BuildTree ($S_2$)
```

(a) Tree building

```
procedure PruneTree (node $N$)
    if $N$ is leaf
        return $C(S) + 1$
    $minCost_1$ := PruneTree ($N_1$)
    $minCost_2$ := PruneTree ($N_2$)
    $minCost_N$ := $\min(C(S) + 1,$
            $C_{split}(N) + 1 + minCost_1 + minCost_2)$
    return $minCost_N$
```

(b) Tree pruning

Figure 1: Algorithms for decision trees

The most time-consuming part of decision tree construction is obviously the splitting point selection. For each active node the subset of data (a *partition*) fulfilling the conjunction of the splitting conditions of the node and its predecessors has to be constructed and for each remaining attribute the possible splits have to be evaluated. Though selecting the best split point based on the measures described above requires no access to the data itself, but only to statistics about the number of records where a combination of attribute value and class label occurs. This information can be obtained from a simple table consisting of the columns *attrib-name*, *attrib-value*, *class-label* and *count*. This structure is described in [CFB99] as *CC table* and in a similar form as *AVC group* (Attribute-Value-Class) in [GRG98]. It could be created using a SQL query of the following kind[CFB99]:

```
select 'A1' as aname, A1 as avalue, C as class, count(*)
from S
where condition
group by A1, C
  union all
select 'A2' as aname, A2 as avalue, C as class, count(*)
from S
where condition
group by A2, C
```

```
    union all
  ...
```

The optimizers of most database systems are usually not able to construct a plan consisting of only a single scan: typically at least for each grouping a separate scan is required. Thus computing the statistics table in a single scan would be a good candidate for a classification primitive as already observed in [CFB99].

An alternative approach would be a primitive that returns directly the split criterion. However, as already mentioned the individual classification algorithms differ in the measure used for choosing the spit. Thus, a primitive which computes only the necessary statistics supports a wider range of algorithms than a specialized primitive.

Considering the queries for selecting the examples belonging to a partition of a node another observation holds. These queries are typically partial-match queries with a condition of the form: $P_1 \wedge P_2 \wedge \cdots \wedge P_m$ where P_i is a predicate $A_j \theta v, \theta \in \{<, >, =, \neq\}$ or $A_j \in V$ and $m \leq n$, where n is the number of attributes. For large data sets where a full scan is too expensive an appropriate access path (index) is required for an efficient evaluation. However, simple one-dimensional indexes are not very useful, because apart from the root node and its direct children all other nodes require multi-dimensional selections. Thus, a second potential primitive for classification is a filtering operation for obtaining the partition of a node by implementing a partial-match query, possibly based on a special index structure. The node statistics primitive could benefit from such a operation because the statistics is computed only for the active partition (Fig. 2).

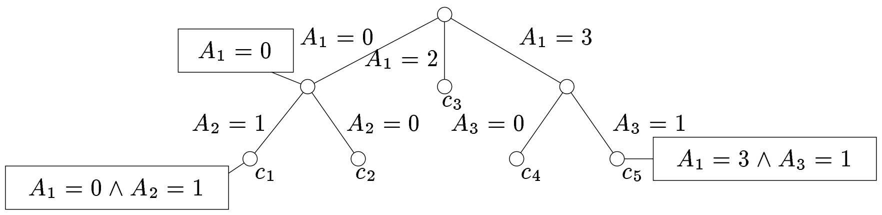


Figure 2: Computing node statistics

In order to prevent over-fitting of the training data, the tree from the growing phase is is pruned by applying the minimum description length (MDL) principle. The basic ideas is that the tree is the best one, which can be encoded using the smallest number of bits. The cost of encoding a tree – called MDL-cost – is computed from ([MRA95])

- the cost of encoding the structure of the tree, e.g. for a binary tree a single bit can specify the kind of a node (leaf, internal node),
- the cost $C_{split}(N)$ of encoding a split at node N that depends on the kind of attribute, i.e. for numeric attributes with v distinct values $\log_2(v-1)$ bits and $\log_2(2^v - 2)$ for categorical attributes,
- the cost $C(S)$ of encoding the classes of records in each node:

$$C(S) = \sum_i n_i \log_2 \frac{n}{n_i} + \frac{c-1}{2} \log_2 \frac{n}{2} + \log_2 \frac{\pi^{\frac{c}{2}}}{\Gamma(\frac{c}{2})}$$

  where S is the set of n records belonging to one of c classes and n_i is the number of records with class label i.

The MDL-pruning is performed by traversing the tree bottom-up and pruning the children of a node N if the cost of minimum-cost subtree rooted at N is greater than or equal to the cost of encoding the records directly at N. The cost of a subtree can be recursively computed. Thus, the most expensive operation during the pruning phase is to compute the cost $C(S)$, which requires information about the number of records belonging to the individual classes in the active partition.

A further task in classification addressed in our work is prediction – applying the induced mining model on new data. This operation is sometimes called *prediction join* because the attribute values of the new data (*the source*) are matched with the possible cases from the model. However, this is not the standard relational join for the following reasons:

- The assignment of class labels to leaf nodes is based on statistical estimations obtained from the training set. Thus, the predicted classes for a given case are annotated by additional statistics, like the probability, which is derived from the training data. In most cases, the prediction is not a single value, but rather a nested table containing classes and probabilities.

- In case of numeric attributes a split is performed by defining a condition of the form $A < v$ for binary splits or $v_1 \leq A \leq v_2$ for n-ary splits. During the prediction join the corresponding bucket for the attribute value of the source data has to be found.

- In order to treat missing values in the source data correctly, aggregating statistics like occurred frequencies are necessary.

Finally, the implementation of the prediction join depends heavily on the model representation. For example, the decision tree could be represented by its nodes and edges as well as the associated conditions and classes or by materializing the individual combinations of attribute values in the nodes (or more precisely the splitting points) together with the predicted class label. Given a particular model presentation a prediction join operator is a further important classification primitive.

3 Primitives: Principle and Implementation

In the previous section we have identified several candidates of primitives for building and applying decision tree classifiers. In this section we describe now these primitives in more detail and present our implementation based on Oracle8i.

Let us first consider the filtering primitive supporting partial match queries. Let $N_i, 0 \leq i \leq n$ be nodes of a decision tree T with N_0 as the root node. With each node $N_i, i \geq 1$ a split condition is associated in form of a predicate P_{N_i} either as $A_i = v$ or as $A_i \in V$. In addition, we assign to each node a class label c_{N_i} that is determined by selecting the most frequent class in the partition of the node. We denote a sequence $N_0 N_1 \ldots N_k, k \leq n$ a decision path, if it holds $\forall i, i \geq 1 : N_i$ is a direct descendant of N_{i-1} in tree T. Each decision path to a node N_p implies a conjunctive condition $\mathit{Cond}_{N_p} = P_{N_1} \wedge \cdots \wedge P_{N_p}$, where each attribute A_i in the predicates occurs at most once. With these definitions we are able to specify the purpose of the filter primitive:

FILTERPARTITION

INPUT:	a condition Cond_{N_p}
	a data set S
OUTPUT:	a partition $S_p \subseteq S$

$$S_p = \sigma_{\mathit{Cond}(N_p)}(S)$$

In principle, several approaches are possible for implementing this filter primitive: multi-dimensional indexes like KdB-tree and others, grid files or bitmap indexes. After some experiments with bitmap indexes supported by Oracle (see section 4 for results), we decided to implement a different approach: multi-dimensional hashing (MDH) [**?**]. MDH is based on linear hashing, where hash values are bit vectors computed for each attribute. These bit vectors are composed to a single value using bit-interleaving (Fig. 3).

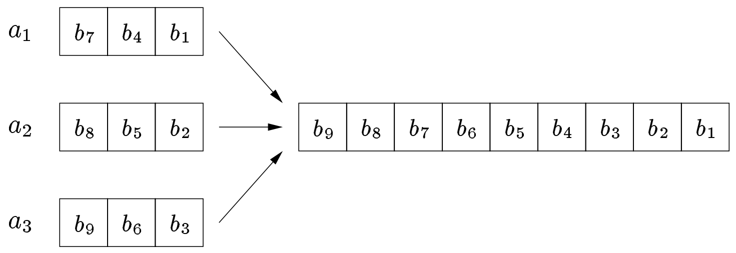


Figure 3: Bit interleaving in MDH

Let $a = (a_1, \ldots, a_n) \in D = dom(A_1) \times \cdots \times dom(A_n)$ a multi-dimensional value, and $b = \sum_{i=0}^{n} |dom(A_i)|$ the number of bits required for encoding the final vector, the composition function $h(a)$ is defined as follows:

$$h(a) = \sum_{i=0}^{b} \left(\frac{a_{(i \bmod k)+1} \bmod 2^{i/k+1} - a_{(i \bmod k)+1} \bmod 2^{i/k}}{2^{i/k}} \right) 2^i$$

As for other hashing schemes the complexity for exact-match queries using MDH is $O(1)$. For partial-match queries it depends on the number of unknown dimensions (attribute values). Let d be the dimensionality and u the number of unknown attribute values, then the complexity is $O(n^{1-\frac{u}{d}})$. Partial-match queries are implemented by setting the corresponding bits in the composed bit vector. For unknown attribute values all combinations of the possible values have to be considered. Therefore, in this case the hash function produces actually a set of hash values.

MDH can be implemented on top of a commercial DBMS in different ways. Probably the most efficient implementation would base on an extensible indexing API as available in Oracle8 or IBM DB2. Here, several routines for index maintenance (insert, delete, update tuples) as well as scan routines (start, fetch, close) are provided by the implementor. In addition, an operator (a SQL function) utilizing the index is defined. This operator could now be used in a query like this:

```
select *
from data
where mdh_match (a1, 1, a2, 2, a3, 0, a4, -1, a5, -1) = 1
```

In this example the arguments for the function `mdh_match` are the attribute values, where -1 denotes an unknown attribute. Thus, the above query is in fact a rewrite of:

```
select *
from data
where a1 = 1 and a2 = 2 and a3 = 0
```

However, due to the complexity of the extensible indexing API we have chosen a different implementation strategy for first experiments, which is – to some extent – more portable to other DBMS. The hash function is implemented as a table function `mdh_partial_match` returning a table of hash values for a partial match query. For the data table an additional column `hid` for storing the hash value derived from the attributes is required. In this way, a join on the values returned by the hash function and the hash values stored in the data table can be performed in order to select the affected tuples. Obviously, an index is required on column `hid` for fast access. Thus, the above query is now formulated as follows:

```
select *
from table (mdh_partial_match (1, 2, 0, -1, -1)) h, data d
where h.id = d.hid
```

For Oracle, an additional cast around the function call and a named table type as result type are required. Moreover, because Oracle supports a collection type `array` a more generic function expecting an array of values as argument can be implemented:

```
create type iarray_t as varray(30) of int;
create type itable_t is table of int;

select *
from table (cast (mdh_partial_match (iarray_t (1, 2, 0, -1, -1)))
     as itable_t) h, data d
where h.id = d.hid
```

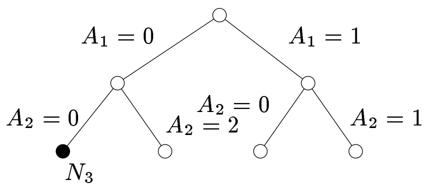
Our approach has several consequences:

- It works only for categorical attributes. Numeric attributes have to be discretized or excluded from indexing.
- For each tuple the hash value has to be computed in advance, e.g. this could be easily done using a trigger.
- In the above described form only conditions of kind $A = v, v \in dom(A)$ are supported.

Whereas the first limitation is inherent to the approach, the third point can be handled by the following extension. For all values of the domain of an attribute an ordering is defined in order to be able to encode each value by a single bit position in a bit vector. Thus, for a condition like $A \in V, V \subseteq dom(A)$ the set V of values are encoded by bitwise disjunction of all values $v \in V$ as given by: $b = \sum_{v \in V} 2^v$. If now the individual values $b_{A_1} \ldots b_{A_n}$ are used as arguments for the function `mdh_partial_match`, the $\in$ condition is supported as well. We evaluate the performance gain of the MDH-based filter operation in section 4.

Filtering the partition associated with a node of the decision tree is only the first part of determining splitting attribute and condition. Both the information entropy and the gini index can be computed from a table summarizing the number of tuples for each combination of an attribute value and a class label. Here, all attributes of the data set not already used as splitting attributes are examined. Therefore, for the partially grown tree shown in Fig. 4(a) and a data set of schema $R(A_1, \ldots, A_n, C)$ at node N_3 the attributes $A_3 \ldots A_n$ have to be considered. Assuming a remaining partition of the data set at N_3 as given in Fig. 4(b) the resulting table contains the information shown in Fig. 4(c).

Let $S_p \subseteq S$ be a partition of the data set S and C_{S_p} the set of class labels occurring in S_p. Furthermore, we define a set of attributes $\mathcal{A} = \{A_1, \ldots, A_m\}$, a set of values $V = \bigcup_i dom(A_i)$ and a



(a) Decision tree

A_3	A_4	A_5	C
0	1	1	c_1
0	1	1	c_1
1	1	2	c_1
2	0	1	c_2

(b) Partition for N_3

aname	*avalue*	*class*	*count*
A_3	0	c_1	2
A_3	1	c_1	1
A_3	2	c_2	1
A_4	1	c_1	3
A_4	0	c_2	1
A_5	1	c_1	2
A_5	1	c_2	1
A_5	2	c_1	1

(c) Node statistics for N_3

Figure 4: Example for node statistics

record $t \in S_p$, where $t(A_i)$ denotes the value of record t for attribute A_i. We specify the COMPUTENODESTATISTICS primitive as follows:

```
COMPUTENODESTATISTICS
    INPUT:      a partition $S_p$
                a set of attributes $\mathcal{A} = \{A_1, \ldots, A_n\}$
                an attribute $A_C \notin \mathcal{A}$ representing the class label
    OUTPUT:     a relation $S_{\text{Stat}} \subseteq \mathcal{A} \times V \times C_{S_p} \times \mathbb{N}$
    It holds
        $(A_i, v, c, count) \in S_{\text{Stat}} \iff$
                $A_i \in \mathcal{A} \wedge v \in V \wedge c \in C_{S_p} \wedge$
                $count = |\{t | t \in S_p \wedge t(A_i) = v \wedge t(A_C) = c\}|$
```

One efficient approach for building this table is to use the super group features introduced in SQL-99 and for example supported by DB2 Version 7. As part of an extended group by clause a list of attribute combinations for creating groups is provided. Thus, the query

```
select
  case
    when grouping(A3) then 'A3' end
    when grouping(A4) then 'A4' end
    when grouping(A5) then 'A5' end
    else NULL
  end as aname,
  case
    when grouping(A3) then A3 end
    when grouping(A4) then A4 end
    when grouping(A5) then A5 end
    else NULL
  end as value,
    C, count(*)
from R
```

```
where condition
group by grouping sets ((A3, C), (A4, C), (A5, C), (C))
```

returns the result table from Fig. 4(c).

An alternative approach is necessary if the advanced grouping features is not supported as in case of Oracle8. We have implemented this feature for Oracle as a table function returning a table of objects of type `avc_group`, which is defined as follows:

```
create type avc_group_t as object (
  aname varchar(30), avalue int, class int, cnt int);

create type avc_table_t is table of avc_group_t;
```

The function `avc_group` requires as arguments a list of attributes which have to be examined, the attribute representing the class label and an input table. This table could be a temporary view as in the following query

```
with pdata as (select * from data where condition)
select *
from table (avc_group ((a1, a2, a3), class, pdata))
```

But because Oracle does not support temporary views and tables as function parameters a complete query string has to be given for the input table in our implementation. Thus, the query looks as follows:

```
select *
from table (cast (avc_group (sarray_t ('a1', 'a2', 'a3'), 'class',
'select * from data where condition') as avc_table_t)
```

In the function `avc_group` the given query is evaluated and the result is processed. The attribute-class-count combinations are collected in a three-dimensional array indexed by attribute name, value and class label. The count values are stored in the fields itself. The cardinalities of the three dimensions could be estimated in advance from the table statistics. In very most cases, this array is small enough to fit into main memory, e.g. for 20 attributes with at most 10 distinct values and 10 classes the size is $20 * 10 * 10 * 4$ Bytes $\approx$ 8 KBytes. After processing the whole result set and collecting the counts the array is used to build the resulting node statistics table, which is finally returned to the caller (Fig. 5).

We will show results of the performance evaluation of the implementation of this primitive in section 4.

As mentioned in section 2 computing the MDL-cost of a subtree is a further candidate for a primitive. In particular, cost of encoding records involves the number of classes, the number of records belonging to an individual class and the total number of records associated with the active node. Obviously, this can be easily combined with the COMPUTENODESTATISTICS, so that both statistics are obtained in one scan. We omit the details here, because of the straight-forward implementation. The cost value computed for a node is stored along with the node in the tree. Based on this, the total MDL cost for each subtree are easily computable during the pruning phase and the pruning can be performed.

After building the decision tree, the induced model can be used to predict the class of new data records, where the class attribute is missing. For this operation – called prediction join – the model has

```
**procedure** COMPUTENODESTATISTICS (query $q$, attribute set $\mathcal{A}$, class attribute $C$)
    initialize array $count$
    execute given query $q$
    **foreach** tuple $t = (a_1, \ldots, a_n, c)$
        **foreach** attribute $A_1, \ldots, A_n \in \mathcal{A}$
            count[$A_i$][$a_i$][$c$] += 1

    **foreach** attribute $A_i$
        **foreach** value $v \in dom(A_i)$
            **foreach** class label $c \in dom(C)$
                **if** $count[A_i][v][c] > 0$
                    produce tuple $(A_i, v, c, count[A_i][v][c])$
```

Figure 5: Algorithm for computing the node statistics

to be interpreted, i.e. by following a path of the tree where for each node the associated split condition is evaluated. We can define the semantics of this operation as follows:

PREDICTIONJOIN

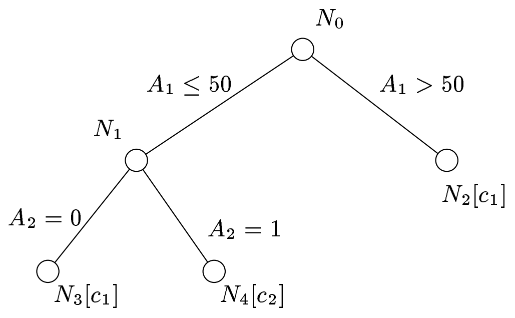
INPUT:	a decision tree T consisting of nodes $N_0 \ldots N_n$
	a source relation $R(A_1, \ldots, A_m)$
OUTPUT:	a relation of predictions $R_P(A_1, \ldots A_m, A_C)$
It holds: $\forall t \in R \quad \exists t_P \in R_P$:	there is a path $N_0 N_1 \ldots N_k, k \leq m \wedge k$ is maximal
	$\wedge \forall i, i = 1 \ldots k : t(A_i) = t_p(A_i) \wedge Cond_{N_p}(t_P) = true \wedge t_P(C) = c_{N_i}$

Because the implementation of the prediction join depends on the model representation, an appropriate structure is required. There is a standard proposal for representing classification tree defined as part of the Predictive Model Markup Language (PMML)[]. However, for storing the tree in a RDBMS a flat table structure is necessary. In Fig. 6 a possible structure is shown together with the corresponding tree. Each tuple in the table represents an edge of the tree from node *parent* to node *node*. Each edge is associated with a condition, where the attribute name is stored in the field *attrib* and the domain for the split is represented by the values of the fields *minval* and *maxval*. The field *class* holds the label of the most frequent class in the partition associated with the node *node* of the edge together with the probability *prob* of the class occurrence.
As the other primitives the prediction join is implemented as a table function. Probably, the best solution would be a function which accepts tables as parameters. But due to the lack of support in current DBMS we have chosen the same approach as already presented for the `avc_group` function: the source and model tables are passed as query strings. A second restriction is the strong typing of the function result. It is not possible to construct a table with a schema that depends on the schemas of the input tables. So, a simple solution is to return a table of a fixed type, e.g. consisting of the primary key of the source table – which is specified as an additional parameter of the function – and the predicted class label.

```
create type pred_t as object (pkey int, class int, probability float);
create type pred_table_t is table of pred_t;
```

The following example shows the usage of the prediction operation.

```
select id, clabel
```



(a) Decision tree

node	*parent*	*attrib*	*minval*	*maxval*	*class*	*prob*
N_0	$\perp$	$\perp$	$\perp$	$\perp$	c_1	0.70
N_1	N_0	A_1	MIN	50	c_1	0.86
N_2	N_0	A_1	50	MAX	c_2	0.66
N_3	N_1	A_2	-1	0	c_1	0.98
N_4	N_1	A_2	0	1	c_2	0.60

(b) Table representation

Figure 6: Representation of a decision tree

```
from table (cast (prediction_join ('model',
   'select id, a1, a2, a3 from source', 'id')) as pred_table_t)
```

The pseudo-code for the prediction join is given in Fig. 7. We assume a model representation as described above (Fig. 6). For each tuple $t_S = (a_1, \ldots, a_m)$ of the source relation the nodes are selected, whose condition is fulfilled by the attribute values of the given tuple. This is performed by the following query $q(t_S)$:

```
select *
from Model
where (aname='$A_1$' and minval < $A_1$ and maxval >= $A_1$) or
      (aname='$A_2$' and minval < $A_2$ and maxval >= $A_2$) or
      ...
      (aname='$A_m$' and minval < $A_m$ and maxval >= $A_m$)
```

These candidate nodes are ordered by their node-id. Next, the candidate nodes are processed in this order as follows: Starting with the root node the next node with a parent-id equal to current node-id is obtained until no further node can be found. In this case, the class and probability values are assigned to the active source tuple. This approach reduces the total number of nodes, which have to be examined.

4 Performance Evaluation

In order to evaluate the performance of our primitives compared to pure SQL queries we performed several experiments. For all tests we used an Oracle8i-DBMS Rel. 8.1.6 runnung on a PentiumIII/500MHz Linux machine with 512 MB of RAM. The primitives were implemented in C as user-defined table functions using the Oracle Call Interface (OCI). The synthetic test data sets were constructed as tables with 10 and 20 attributes and different numbers of tuples up to 100,000. Each attribute contained only discrete values of the interval 0 . . . 3. For the tests of the MDH-based implementation a further attribute containing the hash values was added to the tables. The hash values were computed in advance and an index was created on this attribute.

In the first experiment we studied the performance of different strategies for partial-match queries: a simple full table scan, the usage of bitmap indexes and the MDH-based primitive as described in Section 3.

```
procedure PredictionJoin (source table $S$, model table $M$)
    foreach tuple $t_S = (a_1, \ldots, a_m) \in S$
        execute query $q(t_S)$
        fetch tuple $t_M = (n, p, c, prob)$
        $node := n$
        $class := c$
        $finished :=$ false
        do
            do
                fetch tuple $t_M = (n, p, class, prob)$
                if tuple not found
                    produce result tuple $(a_1, \ldots, a_m, c)$
                    $finished :=$ true
            while $p \neq node$
            $node := n$
            $class := c$
        while $\neg finished$
```

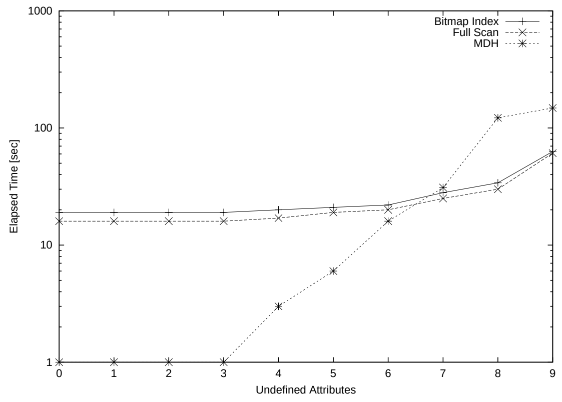
Figure 7: Algorithm for prediction join

Fig. 8(a) shows the running times (for 100 queries) of these strategies for a table of 100,000 tuples with different numbers of undefined attributes. Here, the value 0 at the x axis corresponds to an exact-match query, the value 10 means a full scan in the case of a table consisting of 10 attributes.

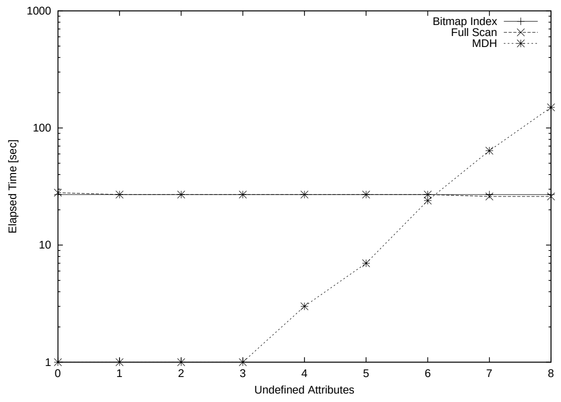
If all or nearly all attributes are given in the query, the MDH approach needs approx. 1.2 seconds for 100 queries, while the scan needs 17 and the access via bitmap indexes needs 19.5 seconds. As expected the elapsed times for the table scans are nearly constant with the growing number of undefined attributes. In contrast, the MDH algorithm only produces approximately similar times for up to 5 undefined dimensions, before the execution times increases. This may be caused by our implementation, because we rely on a traditional b-tree index to find the tuples belonging to the calculated hash values. A second reason could be the inefficient way of returning the table of hash values from the table function, because the table is constructed first and then returned instead of using a iterator-like interface. Fig. 8(b) shows the running times from the experiment with 20 attributes with comparable results.

The difference between the table scan and the bitmap index supported search was of almost no importance in these cases. One conclusion is that the influence of the total number of attributes on the behavior of MDH is not significant. The main factor is the number of unspecified attributes. Here, all test cases produced an excellent performance as long as there are not more than 5 undefined attributes, which corresponds to a selectivity of 50%. However, we believe that a more efficient implementation, e.g. based on the extensible indexing API, could improve this factor.

In the second experiment we evaluated two strategies for computing node statistics: The UnionAll approach (see Section 2 and the AvcGroup primitive. We used the same data sets as described above and compared the running times of queries with different number of grouped attributes. In each query the attributes not involved in grouping were used in the selection condition. As shown in Fig. 9 the running times (again for 100 queries) increased nearly linear with the number of dimensions to be grouped in the case of using UnionAll. For all possible numbers of groups the AvcGroup strategy led to faster



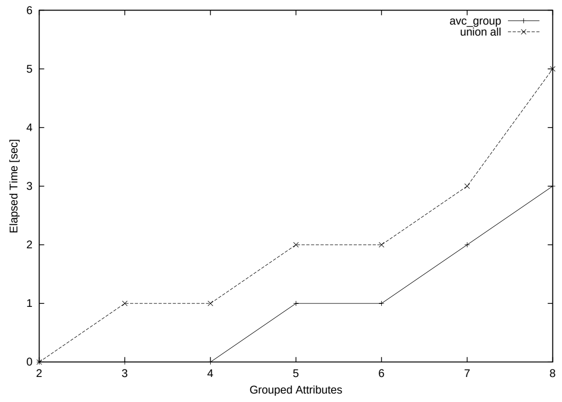
(a) 10 attributes



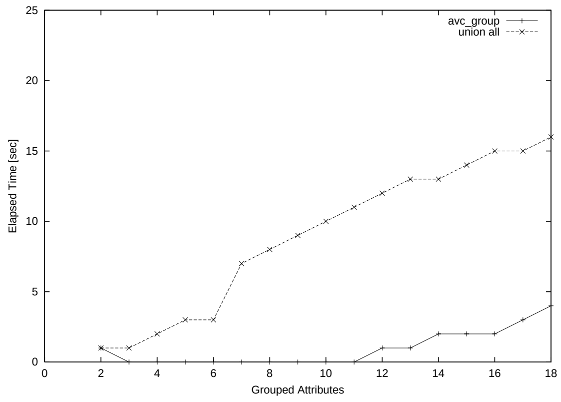
(b) 20 attributes

Figure 8: FILTERPARTITION evaluation results for 100,000 rows

execution. For the data set with 20 attributes similar results were observed.



(a) 10 attributes



(b) 20 attributes

Figure 9: COMPUTENODESTATISTICS evaluation results for 100,000 rows

In the third experiment we studied the effects of the two primitives on a complete decision tree algorithm. For this purpose, we implemented the ID3 algorithm in C++ using SQL queries for filtering the partitions of each node and computing the node statistics. So, no training data has to be hold in memory, only the tree is constructed as main-memory structure. In our implementation no pruning was performed and we considered only categorical attributes. We generated synthetic data sets using the data generator from the IBM Quest project[1] and discretized the numeric values into 4 distinct categorical values. The experiments were performed with data sets of 3,000 up to 50,000 tuples and 9 attributes. We evaluated 4 different strategies of the classifier: SCAN is implemented using pure SQL where the entropy for each attribute is computed by a separate query. For MDH the entropy is computed in the same way

[1]http://www.almaden.ibm.com/quest

(by a separate query), but the filtering of partitions is performed by the MDH primitive if the number of unspecified attributes as less than 5. The UNION strategy is based on the COMPUTENODESTATISTICS primitive, but uses the UNIONALL approach. Finally, the AVC strategy computes the node statistics using the `avc_group` function.

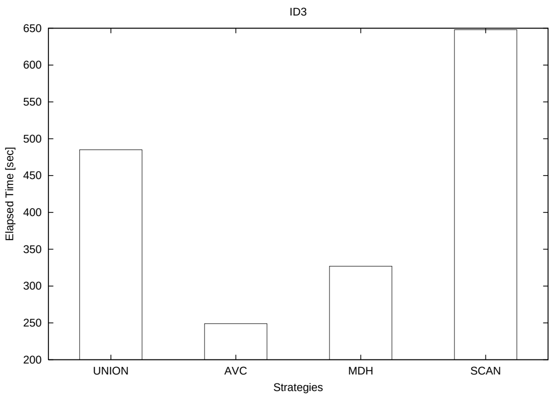


Figure 10: ID3 evaluation results

Fig. 10 shows times for inducing a decision tree from a training set of 50,000 tuples. Both implementations based on the primitives provide a significant mperformance improvement compared with their pure SQL counterparts. Because in both strategies UNION and AVC a scan-based filtering is performed a further improvement could be achieved by using MDH for filtering the partitions in AVC. However, we were not able to evaluate this promising strategy due to the limitations of the current Oracle release, which does not support recursive calls of external functions.

5 Related Work

Many approaches have been proposed to constructing decision tree classifiers in general and particularly to improving scalability of classification. Most well-known algorithm like CART [BFOS84], ID3 [Qui86], C4.5 [Qui93] assume the data to be in memory and therefore are able to work only with relative small data sets efficiently. In the database research scalability is addressed by developing algorithms based on special data structures, e.g. SPRINT [SAM96], SLIQ [MAR96] or optimistic tree construction [GGRL99]. In [GRG98] the RainForest framework is described that introduces an AVC-group data structure providing sufficient statistics for determining the split and algorithms for constructing this structure. [CFB99] describes a similar data structure called CC table and a middleware based on a scheduler ensuring optimized scans and staging. Whereas the RainForest framework does not address SQL databases, the middleware is implemented on a commercial DBMS. Our COMPUTENODESTATICTICS is derived directly from this both approaches. Other approaches consider approximation techniques for scaling up the classification, e.g. sampling [ARS98] and discretization, as well as permitting the user to specify constraints on tree size[ARS98]. Particularly, approximation techniques could be supported by the database systems very well and thus could lead to further primitives.

The integration of data mining and database systems resulting in SQL-aware data mining systems is discussed in [Cha98]. This paper argues for identifying and unbundling a set of new SQL operators or primitives from data mining procedures implementing individual algorithms. Our approach of primitives follows this idea.

Some further examples of tightly coupled data mining techniques are presented in [STA98] for association rules and in [OC00] for clustering. Particularly, for the problem of finding association rules several primitives has been identified and implemented as table functions in DB2. Because of this similar approach we beware that supporting table functions and/or table operators [JM99] is an important issue for implementing data mining primitives efficiently. An ideal supplement to this kind of extension mechanism are user-defined aggregates. An example of a powerful framework for building aggregates and the application in data mining is presented in [WZ00].

6 Conclusions

Tight coupling of data mining and database systems is – beside improving data mining algorithms – a key issue for efficient and scalable data mining in large databases. Tight coupling means not only to link specific data mining algorithms to the database system [HK01], e.g. as stored procedures, but rather that essential data mining primitives supporting several classes of algorithms are provided by the DBMS. Two important tasks convoy the development of such kind of primitives: (1) analyzing data mining functions and identifying common primitives and (2) providing extension mechanisms for an efficient implementation of these primitives as part of the database system API.

In this paper we have presented first results of our work on primitives for decision tree classification. These primitives implement special database operations, which support the SQL-aware implementation of a wide range of classification algorithms. Based on the primitives additional operations are possible. An example are operations for computing the splitting measures (gini index, information entropy), which could be implemented as user-defined aggregation functions for the node statistics table. Moreover, we are convinced that other data mining techniques could benefit from the described primitives as well. For example, partitioning of data sets is a common task and statistics information like node statistics are needed in various mining algorithms. Finally, data preparation as an important preprocessing step of data mining is a further application for databases primitives [**?**].

The experimental results have demonstrated the benefit of the primitives, but also the need for an implementation tighter integrated with the database system. Modern object-relational systems provide already some extension facilities supporting this task (user-defined table functions, user-defined aggregates, extensible indexing). However, our experiences have shown that still more advanced extension APIs are required, e.g. for implementing user-defined table operators which are able to process tables or tuple streams as input parameters. Furthermore, optimizing queries containing this kind of operators is an important but still open issue. Here, techniques considering foreign functions [CS93] or expensive predicates [Hel98] during optimization have to be extended. In our future work, we plan to address this issue. Another task is to exploit the extensible indexing for a more efficient implementation of the multi-dimensional hashing.

References

[ARS98] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A Decision Tree Classifier for Large Datasets. In R. Agrawal, P.E. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of Int. Conf. on Knowledge Discovery in Databases and Data Mining (KDD-98)*, New York City, New York, 1998.

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Chapman & Hall, 1984.

[CFB99] Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Scalable Classification over SQL Databases. In *Proc. of the 15th Int. Conference on Data Engineering (ICDE-99), 23-26 March 1999, Sydney, Australia*, pages 470–479. IEEE Computer Society, 1999.

[Cha98] Surajit Chaudhuri. Data Mining and Database Systems: Where is the Intersection? *Data Engineering Bulletin*, 21(1):4–8, 1998.

[CS93] Surajit Chaudhuri and Kyuseok Shim. Query Optimization in the Presence of Foreign Functions. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 529–542. Morgan Kaufmann, 1993.

[GGRL99] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. BOAT – Optimistic Decision Tree Construction. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 169–180. ACM Press, 1999.

[GRG98] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. RainForest - A Framework for Fast Decision Tree Construction of Large Datasets. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 416–427. Morgan Kaufmann, 1998.

[Hel98] Joseph M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *TODS*, 23(2):113–157, 1998.

[HK01] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman, 2001.

[JM99] Michael Jaedicke and Bernhard Mitschang. User-Defined Table Operators: Enhancing Extensibility for ORDBMS. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 494–505. Morgan Kaufmann, 1999.

[MAR96] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 1996.

[MPC96] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A New SQL-like Operator for Mining Association Rules. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 122–133. Morgan Kaufmann, 1996.

[MRA95] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based Decision Tree Pruning. In U.M. Fayyad and R. Uthurusamy, editors, *Proc. of Int. Conf. on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, 1995.

[NCBF00] Amir Netz, Surajit Chaudhuri, Jeff Bernhardt, and Usama M. Fayyad. Integration of Data Mining with Database Technology. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 719–722. Morgan Kaufmann, 2000.

[OC00] Carlos Ordonez and Paul Cereghini. SQLEM: Fast Clustering in SQL using the EM Algorithm. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 559–570. ACM, 2000.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, (1), 81-106 1986.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[SAM96] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 544–555. Morgan Kaufmann, 1996.

[STA98] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating Mining with Relational Database Systems: Alternatives and Implications. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 343–354. ACM Press, 1998.

[WZ00] Haixun Wang and Carlo Zaniolo. Using SQL to Build New Aggregates and Extenders for Object- Relational Systems. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 166–175. Morgan Kaufmann, 2000.