On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases

Goetz Graefe, Usama Fayyad, and Surajit Chaudhuri

Microsoft Corporation
Redmond, WA 98052, USA
{GoetzG, Fayyad, SurajitC}@microsoft.com

Abstract
For a wide variety of classification algorithms, scalability to large databases can be achieved by observing that most algorithms are driven by a set of sufficient statistics that are significantly smaller than the data. By relying on a SQL backend to compute the sufficient statistics, we leverage the query processing system of SQL databases and avoid the need for moving data to the client. We present a new SQL operator (Unpivot) that enables efficient gathering of statistics with minimal changes to the SQL backend. Our approach results in significant increase in performance without requiring any changes to the physical layout of the data. We show analytically how this approach outperforms an alternative that requires changing in the data layout. We also compare effect of data representation and show that a "dense" representation may be preferred to a "sparse" one, even when the data are fairly sparse.

Introduction
The classification problem is one of the most common operations in data mining. Given a data set of N records with at least m+1 fields: A1, …, Am, and C, the problem is to build a model that predicts the value of the “distinguished” field C (the class) given the values of m fields. C is categorical (otherwise the problem is a regression problem). The class is a variable of interest (e.g., will this customer be interested in a particular product?). The model can be used for prediction of gain understanding of data; e.g. a bank manager may want to find out what distinguishes customers who leave the bank from ones that stay. A classifier is an effective means to summarize the contents of a database and browse it. Applications are many including marketing, manufacturing, telecommunications and science analysis [FU96].

Computational tools from statistics and machine learning communities have not taken into consideration issues of scalability and integration with the DBMS. Most implementations load data into RAM and hence can only be applied on data sets that fit in main memory. The straightforward solution in these settings is to move data (or a sample of it) from the server to the client, then work on this data locally. Moving data has many disadvantages including the cost of crossing the process boundaries and the fact that memory is consumed at the client. Furthermore, such an approach fails to leverage the query processing and other data management functionality of a DBMS. We assume that data is stored in a SQL database, and focus on the problem of supporting a data mining “classification client”. We first note that most familiar selection measures used in classification do not require the entire data set, but only sufficient statistics of the data. We then show that a straightforward implementation for deriving the sufficient statistics on a SQL database results in unacceptably poor performance. Our proposed solution is based on a new SQL operator that performs a transformation on the data on-the-fly and can help a standard SQL backend cope effectively with the problem of extracting sufficient. We demonstrate analytically that our method has better cost performance than other alternatives involving changing the physical layout of data in the database. We also discuss generalizations of the proposed operator.

Preliminaries
Classification algorithms can be de-coupled from data using the notion of sufficient statistics. This suggests a simple architecture (illustrated in Figure 1) in which the database server can support a classification client simply by supplying it with the sufficient statistics, eliminating the need to move data (or copies of it) around.

Greedy Classification Algorithms and Data Access
Widely used prediction algorithms for classification in the data mining literature are decision tree classifiers and the Naïve Bayes classifier [K96]. We limit our attention to discrete data, hence we assume that numeric-valued
attributes have undergone some discretization stage \([B^*84, Q93, F92a, F93, DKS95, MAR96]\). Decision trees are good at dealing with data sets having many dimensions (common in data mining applications), unlike other approaches for classification such as the nearest neighbor [DH73], neural networks, regression-based statistical techniques [PE96], and density estimation-based methods.

Decision trees can also be examined and understood by humans, particularly the leaves viewed as rules [Q93]. Algorithms reported in the literature generate the tree top-down in a greedy fashion. A partition is selected which splits the data into two or more subsets. There are several partition selection measures. Most are based on preferring a partition which results in locally minimizing the class impurity of each subset in the partition \([B^*84]\). Several measures of impurity exist (e.g. Entropy [Q93], the Gini Index \([B^*84]\)).

Other measures outside the impurity family have also been used: the Twoing Rule \([B^*84]\), Gain ratio \([Q86,Q93]\), the J-measure \([GS89]\) and class orthogonality \([F92b]\). The scheme proposed in this paper can support all of the above measures.

**Sufficient Statistics for Selection Measures**

A key insight is that the data is not needed if sufficient statistics are available. We only need a set of counts for the number of co-occurrences of each attribute value with each class variable. Since in classification the number of attribute values is not large (in the hundreds) the size of the counts table is fairly small. Continuous-valued attributes are discretized into a set of intervals. All splitting criteria can be estimated in a straightforward fashion if one had a table, from which we call the correlation counts table \(CC\) (Table 1) for a subset of the data.

The first natural question to ask is how to generate the sufficient statistics using standard SQL? Assuming the data reside in the table DT with \(m+1\) columns \((A1, A2, \ldots, Am, \text{class})\), \(CC\) (Table 1) for a subset of the data satisfying condition “\(\text{Condition}\)”, via SQL code is:

```sql
SELECT "A1" as attrID, A1 as attrValue, class, count(*)
FROM DT
WHERE condition
GROUP By class, attrID, AttrValue
UNION ALL
```

Figure 1. Database server and data mining client

### Problem with the Obvious Approach

The class information entropy of a partition induced by attribute \(A_i\) is defined as:

\[
\text{Entropy}(S|A_i) = \sum_{k=1}^{n} \Pr(A_i = V_j) \text{Entropy}(S_j) .
\]

\([B^*84]\) select the attribute that minimizes this measure. The probabilities are estimated by the frequencies of occurrence of each event in the data subset \(S\).

\[
\text{Entropy}(S|A_i) = -\sum_{j=1}^{n} \frac{\sum_{i=1}^{l} \sum_{k=1}^{l} \text{CC}(i,j,k) \log \left( \frac{\text{CC}(i,j,k)}{\sum_{i=1}^{l} \text{Sum}(i,j)} \right)}{\sum_{k=1}^{l} \text{Sum}(i,j)} .
\]

We can equally well score a partition on a single value versus all other values (binary split) or other splits involving a subset of the attributes \([F94]\). Likewise, many other impurity and statistical measures can be computed strictly from the values in the correlation counts table \(CC\). The same holds true for other measures such as Twoing \([B^*84]\) and Orthogonality \([F92]\).

### Trees for Classification (CC)

Table 1: Sufficient Statistics for Classification \((CC)\)

<table>
<thead>
<tr>
<th>Attribute ID</th>
<th>Attribute Value</th>
<th>Class Value</th>
<th>Count</th>
</tr>
</thead>
</table>

The probabilities are reliably estimate the probability of the class variable using a simple application of Bayes rule: \(\Pr(\text{C} = \text{C}_k | \text{A}_1, \ldots, \text{A}_m) = \frac{\Pr(\text{A}_1 = \text{V}_i, \ldots, \text{A}_m = \text{V}_m) \prod_{i=1}^{m} \Pr(A_i = \text{V}_i | \text{C} = \text{C}_k)}{\Pr(A_1 = \text{V}_1, \ldots, \text{A}_m = \text{V}_m)} \).

The term in the denominator is obtained by summing the terms:

\[
\sum_{i=1}^{k} \Pr(C = C_k) \prod_{i=1}^{m} \Pr(\text{ob}(A_i = \text{V}_i | \text{C} = \text{C}_k) .
\]

The CC table again provides needed terms: \(\Pr(\text{A}_i = \text{V}_j | \text{C} = \text{C}_k) = \text{CC}(i,j,k)\).

#### Sufficient Statistics from SQL Backend

**Problem with the Obvious Approach**

The first natural question to ask is how to generate the sufficient statistics using standard SQL? Assuming the data reside in the table DT with \(m+1\) columns \((A1, A2, \ldots, Am, \text{class})\), \(CC\) (Table 1) for a subset of the data satisfying condition “\(\text{Condition}\)”, via SQL code is:

```sql
SELECT "A1" as attrID, A1 as attrValue, class, count(*)
FROM DT
WHERE condition
GROUP By class, attrID, AttrValue
UNION ALL
```
Select “A2” as attrID, A2 as attrValue, class, count(*)
From DT Where condition Group By class, attrID, attrValue
UNION ALL

... Select “Am” as attrID, Am as attrValue, class, count(*)
From DT Where condition Group By class, attrID, attrValue

The UNION is needed to obtain counts for each of the attributes A1,...,Am with the class. We introduce the condition some_condition to obtain counts for subsets of the data (e.g. individual nodes in a decision tree, etc.)

Most database systems will implement such UNION queries by doing a separate scan for each clause in the union since the optimizers will be unable to harness the commonality among the UNION queries. Observe that the form of the SQL statement using UNION is different from the CUBE operation proposed in [GC*97]. Unlike CUBE, the grouping columns only share the class attribute and no grouping is required for combinations of other attributes. Since we intend to perform such queries over large tables with many columns (hundreds), m scans become quite expensive, we consider methods for avoiding the multiple scans.

New Approach: Unpivoting the Table DT

To avoid the multiple scans caused by the unions, a simple solution could be to change the physical layout of the table. Consider the table DT and suppose each row in DT (each case) were represented as a set of rows, one for each attribute-value pair. We call this table an “unpivoted” form of the table and illustrate it in Table 2:

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Attribute ID</th>
<th>Attribute Value</th>
<th>Class</th>
</tr>
</thead>
</table>

Table 2: Unpivoted form of DT: UDT

Why would the UDT form appear to preferable over DT form for constructing the CC table in SQL? The answer is simple. The query to construct CC, consisting of the union of m sub-queries shown above, reduces to the following simple query over UDT:

Select AttributeID, AttributeValue, class, count(*) From UDT Where condition Group By class, AttributeID, AttributeValue

The reader will note that this query will cause only one scan the table UDT. This is a big win over the m scans required for DT. However, we show next that this win comes at a surprisingly steep increase in the cost of the data scan.

Problem with Materializing Unpivoted Table DT

Consider the scan cost of each of DT and UDT. Let N denote number of cases, m the number of attributes. We assume the cost of storing an attribute value is a constant v bytes. We assume each attribute A_i has value density d_i ≤1.0, i.e. that on average, the proportion of cases over which the A_i has non-NULL value is d_i. We assume that the caseID requires log(N) bits to represent (a lower bound). Similarly, for m attributes, log(m) bits are needed to identify an attribute (a lower bound).

Cost of Scanning DT is: \( N \cdot m \cdot v \) since every attribute value occupies space in DT. Cost of scanning UDT is: \( N \sum_{i=1}^{m} d_i (\log N + \log m + v) \). Difference in scan costs is:

\[
\Delta = N \left( \sum_{i=1}^{m} d_i (\log(Nm) + v) - mv \right).
\]

Assuming same density, then \( \forall i, d_i = d \), hence \( \sum_{i=1}^{m} d_i = md \), yields the simplified form of

\[
\Delta/N = \sum_{i=1}^{m} d_i (\log(Nm) + v) - mv = m[d \log(Nm) + v(d-1)]
\]

(cost per case). We now compare the scan costs of DT and UDT for both dense and sparse data. As we demonstrate below, even for relatively sparse data, the UDT representation incurs a high overhead.

Analysis for Dense Data

When the data is fully dense (d = 1), \( \frac{\Delta}{N} = m \log(Nm) \) which is consistent with our intuition that the DT representation wins by quite a bit. For typical values of N=10^6, m=100, one can see that for a fully dense data set, scanning the UDT table costs over 2000 times more (recall log is base 2) than scanning the corresponding DT table. So for fully dense table, the unpivoted form UDT is not acceptable.

Analysis for Sparse Data

Let us examine the question of when for a classification problem, does it make sense to use the unpivoted representation of UDT over DT. We restrict our attention on the case of a uniform density d (for all i, d_i=d). Examining the form \( \frac{\Delta}{N} = m[d \log(Nm) - v(1-d)] \) yields that the unpivoted representation will only start to win when the expression goes negative. Hence, recalling that \( d \leq 1 \), UDT wins when \( d \log(Nm) < v(1-d) \), which would require that

\[
d < \frac{v}{\log(Nm) + v}.
\]

Note that for large databases this will only occur when the data is at fairly extreme values of densities (for classification, having \( d=0.5 \) is considered quite extreme). Thus, the UDT representation may not be suitable, even if the data are fairly sparse.

Examples

Realistic settings for: size of case ID 3 bytes instead of \( \log(N) \), and size of attr. ID: 2 bytes, and letting \( v = 2 \) bytes, gives cost (case): \( \frac{\Delta}{N} = m[d \log(Nm) - v(1-d)] = 2m(3d - 1) \)

1: Assume \( N=10^6 \), 300 attributes (m=300), and that half the values of all attributes are missing, i.e. \( d=0.5 \), then the difference in scan cost per case is 600(1.5 – 1) = 300. Hence scanning UDT table is equivalent to scanning an extra 300 Megabytes of data! Note that this cost grows linearly with added dimensions and size of original table.

2: For Example 1, assume \( d=0.9 \) (a realistic assumption in typical classification settings) then the difference in scan cost is equivalent to scanning and extra Gigabyte of data. Clearly if data is at an extreme of sparseness, then representation UDT will win, however, such data will not likely be useful for classification algorithms we consider. This leaves us with the conclusion that a traditional SQL
The key to this flexibility is the fact that relational algebra is predicate to a table prior to applying the Unpivot operation: As an algebraic operator, Unpivot can be applied to any language with relational algebra operations in the “from” join employees e on d.dept-id = e.dept-id”. In other words, extensions of the SQL language standard, e.g., the remains closed even after this addition. The proposed syntax produces a table, such as UDT. Thus, relational algebra relational algebra. It consumes a table, such as DT, and it classification applications, is designed to be integrated into the Unpivot operator, as conceived for data mining and derive the desired counts table CC (Table 1).

Incorporating Unpivot operator in Relational Systems

The Unpivot operator, as conceived for data mining and classification applications, is designed to be integrated into relational algebra. It consumes a table, such as DT, and it produces a table, such as UDT. Thus, relational algebra remains closed even after this addition. The proposed syntax is a logical and compatible extension of previous algebraic extensions of the SQL language standard, e.g., the outer join operations in SQL: “select * from departments d left outer join employees e on d.dept-id = e.dept-id”. In other words, there is a well-established precedent for extending the SQL language with relational algebra operations in the “from” clause.

As an algebraic operator, Unpivot can be applied to any intermediate result. For example, in order to apply a search predicate to a table prior to applying the Unpivot operation:

The key to this flexibility is the fact that relational algebra is closed and therefore permits free composition. Efficiency of execution for queries employing the Unpivot operation must be considered. Input rows can be processed one at a time, and the required processing is a mere copying of row values; thus, the implementation code for the Unpivot operator is simple and very fast. Moreover, since one row’s output is completely independent of any other row’s, parallel execution can easily be achieved by partitioning the input rows. In a modern, extensible database query execution engine based on uniform algorithm interfaces [G93], adding a new algorithm relatively easy. Finally, an Unpivot operation affects optimization of neither the query feeding the Unpivot operator’s data stream nor that of the query consuming the Unpivot operation’s output data stream. Thus, index selection based on selectivity and cost estimates are not hampered by the Unpivot operator. In a modern database query optimizer based on algebraic rewrites (e.g. [H89, G95]), the Unpivot operation can be deeply integrated into the query optimizer, and rewrites can be defined that move selection predicates and other operations “up” and “down through” the Unpivot operation. Hence, Unpivot can be integrated into query optimization process as deeply as relational selection and joins are today.

Staggered Applications of UNPIVOT operator

We show how the applications of multiple (two) unpivot operators may be composed for more complex cases of classification when there are either multiple class columns or “attribute” and class columns overlap.

Multiple Class Columns

There could be multiple columns that are to be treated as the “class” columns for CC purposes. If DT had M class columns: (CaseID,A1,A2,…,Am,Class1,…,ClassM), then:

will produce a table with 3+M columns (when M=1 we had 4 resulting columns). Now, we will need to execute a union of M group by operations on the resulting unpivoted table to obtain our desired counts against each of the Class1,…,ClassM columns. However, a clever second application of UNPIVOT will in fact remove the need for the multiple Group By’s. The second UNPIVOT on the output of the first UNPIVOT reduces it down to a table of 5 columns, then a single Group By achieves the desired CC table:

Overlapping “Attributes” and “Class” Columns

The “class” and “attribute” columns may not be mutually exclusive. In general, we would like to treat I of the M columns as “class” columns. For any particular set of counts with respect to any fixed “class column”, say column j, we would like to treat all other M-1 columns as “attributes”. Let DT consists of the following M columns: (A1, A2, …, AM). Assume A1 and A2 are “class” variables, that is, we would like to see the number of times the value of attribute A1 co-occurred with the values of every attribute in A2,…,AM. Same for A2. A simple solution is to preprocess the table DT and replicate the “class” columns as additional
with column replication of A1 and A2 was achieved by the (Select A1,...,Am, A1 AS Class1, A2 AS Class2 from Data_table) statement embedded in the query. There is a potential added scan cost due to column replication, but since the replication is embedded in a pipeline, we assume the table with replicated columns will not be materialized. If replication can be done in the pipeline as an intermediate step, then the technique can be employed to achieve the desired effect efficiently.

Related Work

With proposed extension using Unpivot and architecture of Figure 1, performing classification is reduced to using a simple data mining client that repeatedly queries the database. For decision trees, all the client needs to do is maintain the decision tree structure and implement the scoring functions to be used in selecting the partition at any node in the decision tree. The basic function of the client is to issue a series of queries for requests for counts tables for the nodes in the decision tree being constructed. Interface with the server is straightforward SQL (with our extension). Naïve Bayes is even simpler requiring only one scan. There is a fairly extensive literature on decision tree generation. Since the implementations available for these communities assume the data to be in memory, much of the work has been with very small data sets, typically in the hundreds to thousands of records. Some authors have outlined the challenges of dealing with large data stores [J97]. Much of the work is still focused on virtual memory systems or in-memory caches [ML98].

More recently, data mining and database research has addressed the problem of scaling classification algorithms [J97, MAR96]. The notion of collecting sufficient statistics has been advocated by [J97], but without consideration of issues of integration within the database server. The algorithms in [MAR96] copy the given data set into data structures that correspond to vertical partitions of the table (one partition for each attribute), with at least the class attribute replicated. For high dimensional data, such replication results in increased scan cost. In contrast, we avoid the need to create vertical partitions. An alternative perspective on the Unpivot operator is that it computes all such vertical partitions on the fly by virtue of data transformation in an execution pipeline without materializing the new data view in a single scan over the data table. We also provide a natural, and fairly minimal, extension to SQL. Hence, most of the SQL backend need not be affected. The fact that our scheme does not require copies of the data table such as vertical partitioning makes our approach easier from the point of view of data administration as well.

References


[FU96] U. Fayyad, R. Uthurusamy (Eds), Communications of ACM 39(11), special issue on Data Mining, 1996.


