Building Decision Trees on Records Linked through Key References *

Ke Wang† Yabo Xu‡ Philip S. Yu§ Rong She¶

Abstract

We consider the classification problem where the data is given by a collection of tables related by a hierarchical structure of key references and class labels contained in the root table. Each parent table represents a many-to-many relationship type among its child tables. Such data are frequently found in relational databases, data warehouses, XML data, and biological databases. One solution is joining all tables into a universal table based on the recorded relationships, but it suffers from a significant blowup caused by many-to-many relationships. Another solution is treating the problem as relational learning, at the cost of increased complexity and degraded performance. We propose a novel method that builds exactly the same decision tree classifier as built from the joined table, but not the blowup required in the traditional approach.

1 Introduction

Classification has been identified as an important problem in data mining. A *training set* consists of examples with the class label. Each example represents a real world entity described by several attributes. The objective is to build a *classifier* for determining the class label of unclassified records. This problem has a wide range of applications, including fraud detection, finding buyers, medical diagnosis. Most works focus on identically structured records stored in a single table. However, real life data are stored in differently structured tables that are semantically linked via different types of relationships, as illustrated by the following example.

EXAMPLE 1.1. Suppose that we like to classify the class label "Student satisfaction" in the table T(teaches), where T contains many-to-many "teaches" relationships among professors P (100 records), students S (10,000 records) and courses C (100 records), by containing their *primary keys* as *foreign keys*. Each record in T references *exactly one* record from each of P, S and C tables. This problem is conceptually equivalent to the classification on the joined table by substituting foreign key references with actual records. However, the joined table suffers from a significant blowup due to the many-to-many "teaches" relationship: if each student takes 4 courses, the joined table would contain 40,000 records, in which each S record is replicated 4 times, and each C and P record is replicated 400 times. ■

In this example, the central table contains the class labels and many-to-many relationships among other tables through simultaneous foreign key references to them. We call the classification problem for such databases the *star join classification*. Relational databases designed by the ER model [3] have this structure, so are the star databases widely adopted in the OLAP environment [2], such as the TPC-H benchmark [10]. Foreign key references are also widely used in the DTD of XML documents in the form of nested Element structures, and in biological databases that cross-reference multiple sources.

The above example illustrates two points. First, the star join classification has an equivalent single table representation, namely, the joined table, thus, a classic solution based on the joined table. Recognizing this fact would enable the performance (i.e., accuracy) guarantee of classic solutions. Second, the many-to-many relationships cause a blowup in the joined table and a significant overhead to the subsequent classifier construction. Thus, obtaining a classic solution by applying a classic method to the joined table is not scalable.

We propose a scalable decision tree algorithm for the star join classification. Our insight is that the operation at each decision tree node only depends on the class frequency of attribute values, not the

*Research was supported in part by research grants from the Natural Science and Engineering Research Council of Canada

†Simon Fraser University, wangk@cs.sfu.ca
‡Simon Fraser University, yxu@cs.sfu.ca
§IBM T. J. Watson Research Center, psyu@us.ibm.com
¶Simon Fraser University, rshe@cs.sfu.ca

availability of the joined table. Thus, we can propagate the class label to all tables T, and at a decision tree node we split each table T *individually* to get exactly the same set of records at each child node as if the joined table was split. In this method, for each record in T, all its occurrences in the joined table are now represented by a single occurrence plus the class count matrix representing the aggregated class count of those occurrences. Therefore, we are able to compute the same split attribute as using the joined table. The strategy for efficiency is *counting* occurrences instead of *duplicating* records.

In Section 2, we define the problem and review related works. In Section 3 we present our approach. Finally, we conclude the paper.

2 Problems and Related Works

2.1 Star join classification

A *star database* [2] is a rooted tree of tables. T is the *fact table* (the parent) of a *dimension table* A (a child) if T contains the primary key A_ID of A as a foreign key. In this paper, we consider the classification where the class label is contained in the root table. Each example consists of the records involved in a relationship in the root table. In other words, each example is represented by a record in the joined table obtained using foreign key references. There is an one-to-one correspondence between the records in the root table and the records in the joined table.

DEFINITION 2.1. (STAR JOIN CLASSIFICATION) Given a star database in which the root table contains the class label, build a classification model with a cost measured by the size of the star database (not by the size of the joined table). ■

2.2 Decision tree based classification

We consider the decision tree based classification. A classic description of decision tree construction can be found in [1, 7]. The decision tree is built in two phases. In the *top-down construction phase*, shown in Algorithm 1, the table is recursively partitioned until each partition consists of entirely or dominantly examples from one class. At each node n of the decision tree, there are two steps:

Find the split attribute (Line 4). This step selects the split attribute and criterion to maximize the skewness of class distribution. The essential information needed is the count matrix, or termed as *AVC sets* (for Attribute-Value-Class) [5]. The AVC set for an attribute Att at a node n, denoted $AVC_n(Att)$, contains

Algorithm 1 Top-down decision tree construction

$BuildTree(n)$
Input: node n
Output: decision tree rooted at node n

```
1: if the stopping criterion is met then
2:    return;
3: end if;
4: Crit=the split criterion at n;
5: T=the database at n;
6: create child nodes n1 and n2;
7: SPLIT(n, T, Crit);
8: BuildTree(n1);
9: BuildTree(n2);
```

the class frequency for each distinct value v in Att. For a numerical attribute, $AVC_n(Att)$ is sorted in the order of values of Att. As in [9, 5], we consider the binary split criterion where the table at a node n is split between two child nodes n_1 and n_2. The detail can be found in [9, 5].

Partition the database (Line 7). This step splits the table at the parent node n between the child nodes n_1 and n_2 according to the split criterion. It reads the table at n and writes the tables at n_1 and n_2 to disk. In the same scan, it also collects $AVC_{n_i}(Att)$. $AVC_{n_i}(Att)$ has a size proportional to the number of distinct values in Att, not the number of records. For the depth-first construction of the decision tree, the memory is only required to hold the AVC sets for the two child nodes being created.

In the *bottom-up pruning phase*, some subtrees or branches are pruned to ensure that the model does not over-fit the training set. This phase uses only the class frequency at leaf nodes, not data records. Since our algorithm produces the same class frequency as in the decision tree built from the joined table, this phase makes no difference to our algorithm.

2.3 Related work

CrossMine [11] presented a scalable relational learning. We consider the star join classification, which has a classic solution based on the joined table, therefore, the performance guarantee of the well researched decision tree classification. [11] propagates the IDs of target records along the search path to avoid expensive joins. We propagate the class label, instead of IDs of target records. The former has a size equal to the number of distinct classes, whereas the latter does not have a pre-determined size. We search the split attribute from all tables, whereas [11] restricts the search of predicates to "active" tables related to the rule. We

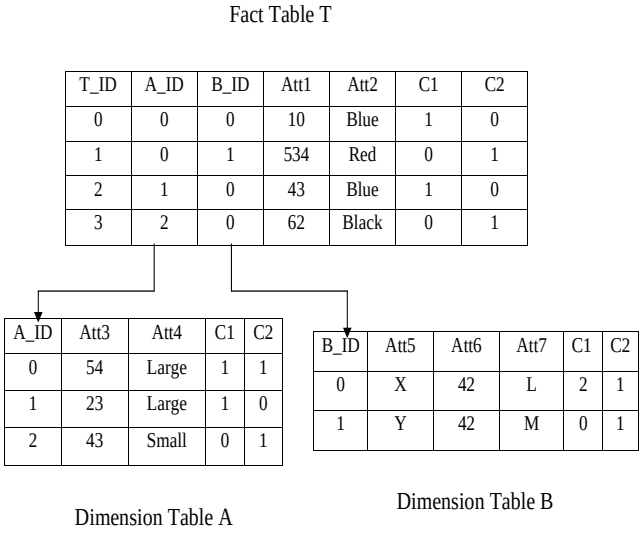

Figure 1. The running example

consider disk-resident databases whereas [11] considers in-memory databases.

Scalable decision tree construction for a single table has been studied in the database field [5, 4, 6, 8, 9]. The focus of these works is scaling up the construction for databases stored on disk. These methods, no matter how scalable, suffer from the initial blowup of the joined table. This remark applies to the sampling approach [4] where the full joined table needs to be examined to build the *exact* decision tree. Sampling methods that do not examine the full joined table are not guaranteed to produce the exact decision tree.

3 Our Approach

Our algorithm, Star_DT, has two main ideas that make it scalable. Before the decision tree is constructed, it propagates the class label from the root table to all tables so that no join is needed to get the class label afterwards. At each decision tree node, it selects the split attribute by evaluating attributes within their own tables without join, and splits individual tables, instead of the joined table, to ensure that each table contains exactly the same information over its attributes as contained in the joined table. Below, we explain these ideas in details.

3.1 Propagating the class label

Before constructing the decision tree, we first propagate the class information to all tables. For each table, the new columns $C_1, \cdots, C_k$, one for each class C_i, store the class count. For the root table, $C_i = 1$ if the record has the class C_i, and $C_i = 0$ otherwise. Recursively, we propagate C_i from a fact table T to a dimension table A as follows: C_i for each record $A\_ID = j$ in A is equal to $SUM(C_i)$ over the records in T that have the foreign key value $A\_ID = j$. The records with the all-zero class count are removed from A. This propagation ensures that, for each record t in a table, C_i is equal to the sum of C_i of the records in the joined table that agree with t on the attributes of t.

EXAMPLE 3.1. (THE RUNNING EXAMPLE) In Figure 1, A and B are dimension tables of T. A_ID and B_ID are primary keys of A and B and foreign keys in T. $Att1$, $Att3$, $Att6$ are numerical attributes and others are categorical attributes. B contains the new columns C_1 and C_2. For the record with the primary key value $B\_ID = 0$, $C_1 = 2$ and $C_2 = 1$ because three records in T reference this record, two having the class C_1 and one having the class C_2. ■

Subsequently, at each decision tree node, we find the split attribute and criterion by computing AVC sets using the class count matrix $C_1, \cdots, C_k$ in each table, and split each table by "propagating" the split criterion. We explain these steps below.

3.2 Finding the split attribute

Let $Join(n)$ denote the partition at a decision tree node n of the joined table. Note that we will not construct $Join(n)$. Let $Star(n)$ denote the partition at n of the star database. The following theorem shows that each table T in $Star(n)$ is a projection of $Join(n)$ with duplicates represented by a single record and an aggregated class count matrix. Therefore, if we compute the AVC sets using $C_1, \cdots, C_k$ in the table T, we have exactly the same result as computed using $Join(n)$. No join is needed.

THEOREM 3.1. Consider a table T in $Star(n)$. Let $Schema(T)$ be the set of attributes in T (excluding the new class columns). T is equal to

SELECT $Schema(T)$, $SUM(C_1), \cdots, SUM(C_k)$
FROM $Join(n)$
GROUP BY $Schema(T)$ ■

The proof of Theorem 3.1 for the decision tree root follows from our class propagation. For other nodes, the proof follows from our splitting of $Star(n)$ among child nodes.

3.3 Splitting a table

Suppose that a node n is split into two child nodes n_1 and n_2. We like to obtain $Star(n_i)$ by splitting

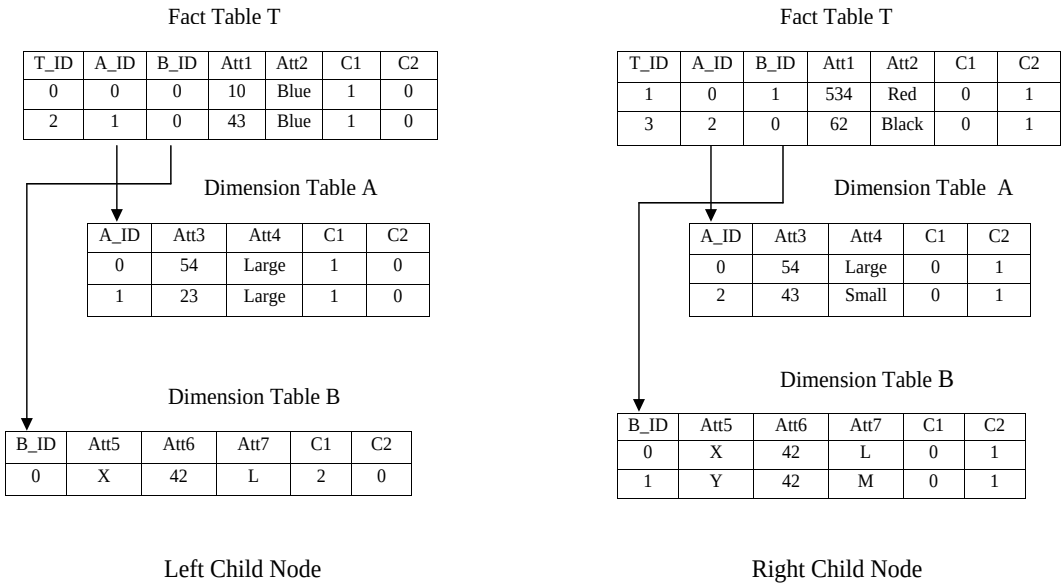

Figure 2. Splitting tables by $Att1 < 52.5$

every table in $Star(n)$ between the two child nodes. While splitting a table, the AVC sets at the child nodes are collected in the same scan. First, we split the "seed" table containing the split attribute as usual. Recursively, we propagate the splitting to dimension tables and fact tables based on foreign key references. The propagation assumes that $Star(n)$ is stored on disk.

3.3.1 The fact-to-dimension propagation Suppose that we know how to split a table T in $Star(n)$. To propagate the splitting to a dimension table A of T, while splitting T we "index" the splitting of foreign key values on A_ID as described below.

DEFINITION 3.1. (F2D INDEX) Consider a decision tree node n. Let A and T be tables in $Star(n)$ and A be a dimension table of T. Suppose that we know how to split T. For each record $A\_ID = i$ in A and each child node c of n, the *Fact-to-Dimension index (F2D index)* for A contains an entry denoted $F2D_A(i, c)$. $F2D_A(i, c)$ stores the aggregated class count matrix over all records in T that reference the record $A\_ID = i$ and split to the child node c. ■

To create the $F2D_A$ index, while splitting T, if a record t splits to a child node c, we increment $F2D_A(i, c)$ by $< t[C_1], \cdots, t[C_k] >$, where i is the foreign key value on A_ID in t. Intuitively, $F2D_A(i, c)$ gives the portion of the class count matrix in the record $A\_ID = i$ that splits to the child node c. We use this information to split the records in A.

EXAMPLE 3.2. Consider Figure 1 again. Suppose that T is split by the split criterion $Att1 < 52.5$, the midpoint between 43 and 62. Figure 2 shows the split T. Consider splitting the record $B\_ID = 0$ in B. In T, the records $T\_ID = 0$ and $T\_ID = 2$ reference the record $B\_ID = 0$ and split to the left child node. So, the record $B\_ID = 0$ splits to the left child node with the aggregated class count matrix of the records $T\_ID = 0$ and $T\_ID = 2$, i.e., $< 2, 0 >$ (for C_1, C_2 in that order). Similarly, the record $B\_ID = 0$ splits to the right child node with the class count matrix $< 0, 1 >$. The effect is that the record $B\_ID = 0$ splits its class count matrix $< 2, 1 >$ between the two child nodes. ■

The F2D method. Assume that $F2D_A$ is available (thus, the fact table of A was split). We split A as follows. For each record r in A with the primary key value i, if $F2D_A(i, c)$ is not all-zero for a child node c:

- r splits to the child node c with the class count matrix $C_1, \cdots C_k$ given by $F2D_A(i, c)$;
- for a (unsplit) dimension table B of A (if any), increment $F2D_B(j, c)$ by $F2D_A(i, c)$, where j is the foreign key value on B_ID in r.

3.3.2 The dimension-to-fact propagation To propagate the splitting from a dimension table A to its fact table T, while splitting A we index the splitting of its primary key values as described below.

DEFINITION 3.2. (D2F INDEX) Consider a decision tree node n. Let A be a dimension table in $Star(n)$.

Suppose that we know how to split A. For each record $A\_ID = i$ in A, the *Dimension-to-Fact index (D2F index)* for A contains an entry denoted $D2F_A(i)$. $D2F_A(i)$ stores the child node to which the record $A\_ID = i$ in A splits. ■

To create the $D2F_A$ index, while splitting A, for each record that has the primary key value i and splits to a child node c, set $D2F_A(i) = c$. Intuitively, $D2F_A(i)$ contains the child node for all the records in the fact table of A that contain the foreign key value $A\_ID = i$.

The D2F method. Assume that $D2F_A$ is available (thus, A has the fact table). We split the fact table T of A as follows. For each record t in T having the foreign key value i on A_ID and the primary key value i', let c be the child node given by $D2F_A(i)$:

- t splits to the child node c;
- if T has an unsplit dimension table B: increment $F2D_B(j, c)$ by $< t[C_1], \cdots, t[C_k] >$, where j is the foreign key value on B_ID in t,
- if T has an unsplit fact table (only if T is not the root table), set $D2F_T(i')$ to c.

EXAMPLE 3.3. Suppose that the split criterion at the root node is $Att3 < 33$, i.e., the midpoint of 23 and 43 in A. First, we split A using the split criterion, and build $D2F_A$. Thus, the record $A\_ID = 1$ splits to the left child node, and the records $A\_ID = 0$ and $A\_ID = 2$ split to the right child node. And, $D2F_A(0) = D2F_A(2) = Right$ and $D2F_A(1) = Left$.

Next, we split T using the $D2F_A$ method, and build $F2D_B$: for each record t in T, we look up $D2F_A$ by $t[A\_ID]$ to find the child node c for t, and increment $F2D_B(t[B\_ID], c)$ by $< t[C_1], \cdots, t[C_k] >$. Thus, the third record in T splits to the left child node because $t[A\_ID] = 1$ and the other three records in T split to the right child node. One can verify that $F2D_B(0, Left) =< 1, 0 >$, $F2D_B(1, Left) =< 0, 0 >$, $F2D_B(0, Right) =< 1, 1 >$, and $F2D_B(1, Right) =< 0, 1 >$.

Finally, we split B using the $F2D_B$ method. We omit this part because it is similar to Example 3.2. ■

It is worth noting that for both $F2D_A$ and $D2F_A$, A must be a dimension table. This implies that we never create an index that has a size proportional to the cardinality of the root table. Therefore, even if we keep all indexes (at a decision tree node) in memory, they still use less space than the hash table for joining attribute lists in Sprint [9].

3.4 Analysis

The dominating work at each iteration is propagating the splitting among the tables T in $Star(n)$ for splitting the current decision tree node n. This work essentially performs the matching as required for computing $Join(n)$. However, it does not materialize the join result, but only computes the class count matrix for each distinct record in the projection $\prod_T(Join(n))$ (Theorem 3.1). Thus, while $Join(n)$ duplicates each record in T as many times as it occurs in $Join(n)$, as is typically the case because each fact table represents a many-to-many relationship type among its dimension tables, $Star(n)$ simply keeps a count for each class label to represent the duplicates. This counting strategy speeds up the data scan at each decision tree node. Due to the space limitation, we have to report experimental results elsewhere.

References

[1] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and regression trees.* Wadsworth: Belmont, 1984.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[3] P. P.-S. Chen. The entity relatoinship model - towards a unified view of data. *ACM TODS*, 1(1):9–36, March 1976.

[4] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. Boat - optimistic decision tree construction. In *SIGMOD*, 1999.

[5] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB*, 1998.

[6] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: a fast scalable classifier for data mining. In *EDBT*, 1996.

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

[8] R. Rastogi and K. Shim. Public: A decision tree classifier that integrates building and pruning. In *VLDB*, 1998.

[9] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *VLDB*, 1996.

[10] TPC. Tpc benchmark h standard specification. In *http://www.tpc.org/tpch/spec/tpch2.0.0.pdf.*

[11] X. Yi, J. Han, J. Yang, and P. Yu. Crossmine: efficient classification across multiple database relations. In *ICDE*, 2004.