Attribute Metadata for Relational OLAP and Data Mining

T. H. Merrett
McGill University, Montreal, Canada

May 27, 2002

[1]

Abstract

To build the d-dimensional *datacube*, for on-line analytical processing, in the relational algebra, the database programming language must support a loop of d steps. Each step of the loop involves a different attribute of the data relation being cubed, so the language must support attribute metadata. A set of attribute names is a relation on the new data type, **attribute**. It can be used in projection lists and in other syntactical postions requiring sets of attributes. It can also be used in nested relations, and the **transpose** operator is a handy way to create such nested metadata. Nested relations of attribute names enable us to build decision trees for classification data mining. This paper uses OLAP and data mining to illustrate the advantages for the relational algebra of adding the metadata type **attribute** and the **transpose** operator.

Keywords relational algebra, datacube, data mining, association, classification, decision trees, nested relations, metadata

1 Introduction

In introducing the term "on-line analytical processing" (OLAP), Codd [2] wrote

> ..relational DBMS were never intended to provide the very powerful functions for data synthesis, analysis and consolidation that is being defined as multi-dimensional data analysis

and this is certainly true of SQL and current commercial database systems [4]. We will show that a database programming language which thoroughly integrates programming language and relational database concepts can be used to build any specific datacube without arbitrary extensions or operators.

The classical relational algebra shows limitations, as Codd says, when it tries to generalize this implementation to arbitrary datacubes, because it has no facility to loop over the attributes of a relation. A simple adaptation of the relational algebra to permit this—it is hardly an extension—is to allow *metadata* of type **attribute** among the legal attribute types (such as **integer**, **boolean**, **string**, etc.).

Non-first-normal form, or *nested* relations [6], have been discussed almost as long as Codd's original "flat" relations, with considerable emphasis on special algebras and operators to convert between nested and flat relations (e.g., [5, 3], etc.). If the classical relational algebra is complemented by a "domain algebra", which is useful in many other ways [7], the only new syntactic construct needed to express and work with nested relations is a mechanism which forms a collection of attributes; the rest is provided by subsuming the relational algebra into the domain algebra.

To build a decision tree, say for classification data mining, we need to choose an attribute using information theory, examine the values for this attribute, and then, for each value, repeat the cycle with other attributes. This requires that we have information about attributes and their values, both available at the same level, and both accessible to the domain and relational algebras. We introduce an operator to create a nested relation which forms the "transpose" of the attribute-value pairs in each tuple. This transpose has two attributes, one of which is, of course, of type **attribute**.

In this paper, we review the domain algebra, mainly via the application of constructing specific datacubes. In section 3, we introduce **attribute** metadata, and go on to apply it to building general datacubes and to start building decision trees. We then review nested relation operations as an extension of the domain algebra, and apply them to association data mining. Section 5 describes the new **transpose** operator of the domain algebra, and uses it to construct decision trees for classification data mining.

[1] Copyright for non-electronic reproductions ©Springer-Verlag, 2002. Electronic and executable forms of this work are copyleft ©T. H. Merrett, 2002

2 The Domain Algebra

The domain algebra has been around for a couple of decades, but not widely used, which has had unfortunately expensive consequences in terms of programming effort and confusion. It is thus appropriate to review it briefly here, saving illustrative examples for section 2.2, which uses it to build a datacube.

The algebra (which should have been named "attribute algebra") constructs new attributes from existing attributes in two ways. *Scalar* operations work "horizontally" (in terms of the usual table representation of relations) along the tuples. *Aggregation* operators work "vertically" along attributes.

All the normal calculations we might expect to be able to do on *scalar* data have corresponding scalar operators in the domain algebra: arithmetic, logical operations, string operations, the usual mathematical functions, and conditional expressions. Here are some examples, where the variables (*N, P, Windy*, etc.) are attributes. Each of these examples will be used later in the paper.

$N + P$
sum1/*sum2*
$(N + P) * \lg(N + P) - N \lg(N) - P \lg(P)$
if *Windy*=`"ANY"` **then** 0 **else** N

A domain algebra statement is distinguished from a relational algebra statement syntactically, and it is helpful to give that syntax here. The attribute resulting from the first expression, above, can be named *np* using

let *np* **be** $N + P$;

Special uses of the scalar domain algebra are renaming

let N **be** *totN*;

and creating constant attributes

let *seq* **be** 0;

or

let *Windy* **be** `"ANY"`;

We will omit the **let .. be** in the remaining examples of this subsection, showing only domain algebra expressions.

The *aggregation* operations of the domain algebra are of two kinds, *reduction*, and *functional mapping*, and these each break into two subcategories. We consider only reduction here, and its subcategories, *simple reduction* and *equivalence reduction*. These generalize the five aggregation functions of SQL, which are written

red + **of** A	// sum values of attribute A
red + **of** 1	// count
red min of A	// find least value of attribute A
red max of A	// find greatest value of attribute A

and the average is a scalar division of two aggregations

(**red** + **of** A)/**red** + **of** 1

The same statistics can be calculated for groups of tuples that form equivalence classes determined by having the same value for a specified attribute or set of attributes, say G, H.

equiv + **of** A **by** G, H
equiv + **of** 1 **by** G, H
equiv min of A **by** G, H
equiv max of A **by** G, H

An average we shall make use of in building a decision tree is

(**equiv** + **of** *npinp* **by** *Outlook, Humidity*)/
equiv + **of** *np* **by** *Outlook, Humidity*

The advantage of this notation is that it can be used to express any aggregate or statistic we can define, such as a standard deviation, without limiting the programmer to a pre-defined set of functions. Not only +, **min** and **max** can be used, but any associative and commutative operator, such as multiplication or, as we shall see later, the natural and outer joins of the relational algebra.

2.1 Actualization: Relational Algebra

Apart from the improved flexibility over SQL, the foregoing domain algebra seems pretty trivial. It involves, however, a significant subtlety, which is responsible for the considerable intellectual simplification it permits in tackling complex problems. This is that every domain algebra operation is completely independent of relations; no relation is referred to in any expression of the domain algebra. The attributes produced by the domain algebra are *virtual*. So, of course, to populate them with data, they must be *actualized* in the context of some relation or other. This can be done by the relational algebra, with no syntactic extensions.

It may be helpful to think of a domain algebra atatement as the definition of a parameterless function, whose name is the identifier following **let** and whose body is the expression following **be**. No execution results when the language interpreter encounters the domain algebra statement; execution is postponed until the newly defined attribute is used somewhere in the relational algebra.

The advantage of the independence of the domain algebra from relations is that we can think separately about the two aspects of any problem which involves both relational operations and calculations on the attributes. It also enables us to implement the domain algebra in any syntax which already contains relational operations, such as SQL, without syntactic modifications.

However, anticipating the need in nested relations for relational operators, such as natural join, to appear explicitly in **red** and **equiv** expressions of the domain algebra, we are going to have to alter the formulation of SQL. We discuss this now, because the relational algebra is, as we said, the context in which virtual attributes are actualized. We need a syntax which makes explicit the unary and binary operators of the relational algebra, and which clearly distinguishes expressions from statements. We call this "programming notation", to distinguish it from

SQL which, as a query language, was never intended for programming.

The most straightforward means of actualizing a virtual attribute created by the domain algebra is through *projection*. Here is an example, using first SQL then programming notation.

Suppose we have a relation

Training(*Outlook, Humidity, Windy, N*)

and the domain algebra

let *totN* **be equiv** + **of** *N*
by *Outlook, Humidity*;

Then the sum can be actualized, together with the other relevant attributes, using

select *Outlook, Humidity, totN*
from *Training*

The programming notation for the projection is almost the same

[*Outlook, Humidity, totN*] **in**
Training;

and there is no apparent motivation for the change. But let us go on to the next step we will have to take in the next section, namely to rename *totN* back to *N*.

let *totN* **be equiv** + **of** *N*
by *Outlook, Humidity*;
let *N* **be** *totN*;

We need two projections, one after the other, because if we try to actualize the virtual *N* in a relation which already has an actual *N* there would be ambiguity in the result. In SQL, we would have to write this

select *Outlook, Humidity, Windy, N*
from
select *Outlook, Humidity, totN*
from *Training*;

while programming notation is tidier

[*Outlook, Humidity, Windy, N*] **in**
[*Outlook, Humidity, totN*] **in** *Training*;

So far, the notational differences are not a big deal. Let us go on to *selection*, which SQL writes

select <attribute list>
from <relation>
where <tuple conditional expression>

and programming notation writes

[<attribute list>]
where <tuple conditional expression>
in <relational expression>

This rearrangement of the SQL order puts the <relational expression> last so that compound expressions are more easily further articulated.

We proceed to *binary* operators, the joins, which programming notation writes as infix expressions of explicit operators. This also allows compound expressions to be built up, and several types of join to be defined. We use three in this paper, extensions to relations of the set operations of intersection, union, and difference.

Programming notation is

<relational expression> <join operator>
<relational expression>

or, if the join attributes must be named explicitly because their names are not common to the two operands,

<relational expression>
[<attribute list> <join operator>
<attribute list>]
<relational expression>

The natural join generalizes set intersection, and we write the operator as **natjoin**. The similar generalization of set union gives the outer join, which we call **union**. Finally, set difference becomes **diff**, the difference join. Since in the paper we mainly use these latter two as ordinary set operators, we do not explain the extensions, which require null values in the case of **union**.

We can discuss the difference in notation in terms of a natural join between a relation *ShoppingBaskets*(*xact, item*) and a relation *ShoppingBaskets'*(*xact, item'*). (This pairs up all items that share the same transaction in an association-mining implementation.) SQL uses an expression which closely resembles its unary operators,

select *
from *ShoppingBaskets*, *ShoppingBaskets'*
where
ShoppingBaskets.xact=
ShoppingBaskets'.xact

While this makes explicit that equality across relations between the attributes of the names, *xact* and *xact*, is tested, it does not make explicit the higher-level idea that a natural join is being described. The programming notation,

ShoppingBaskets **natjoin** *ShoppingBaskets'*

uses the **natjoin** operator so that we can explicitly ascribe all the known properties of the natural join to this expression.

The apparent greater generality of SQL joins is only apparent: programming notation can always use a selection operator to change the join condition from equality to something else. On the other hand, useful operations such as outer and difference joins cannot be expressed using only the above SQL.

We will also use a fourth join, **comp**, the natural composition, which is a natural join followed by projecting only the non-join attributes. For example,

ShoppingBaskets **comp** *ShoppingBaskets'*

would be translated into the above SQL, modified only in the first line, which becomes

select *item, item'*

Comp deserves a special name, rather than being implemented just as a join followed by a projection, because it is frequently used to join a relation with itself on different attributes from the two sides, so that cumbersome attribute renaming would be required by the join-project implementation.

Comp actually belongs to a separate family of relational joins from the three others. This family extends the set-*comparison* operators (subset, superset, set equality) to relations, and includes the **division** operator, which we do not use in this paper. **Comp** corresponds to the test for non-empty intersection of two sets.

Comp and the other joins in this family exclude the join attributes from the result, and raise the possibility (when they are used as set comparisons) that the result may have *no* attributes. We now assert that a *nullary* relation serves as a Boolean, with the empty state interpreted as `false` and the non-empty state as `true`. We can therefore also use the nullary projection,

[] **in** <relational expression>

which is true if the value of the relational expression contains any tuples, and can be pronounced "something in <relational expression>".

So far, we have looked at *expressions* only of the relational algebra. They may be combined into compound expressions of arbitrary complexity (although it is not usually practical to write more than a join or two, and a couple of selection-projections in a single expression). We must say how relational expressions can be written into statements of the programming language. There are two kinds of statement.

The *assignment* statement creates a new relation, or overwrites an old one, from the value of a relational expression, just as assignment statements in any imperative programming language do. The syntax is

<identifier><− <relational expression>;

The domain algebra and all the relational algebra operators described so far, except for the assignment operator, are *functional*, in the technical sense that the same operands always give the same results, and there are no side-effects. Functional programming is very elegant and avoids most of the opportunities a programmer has for error in non-functional languages. Database programmming, however, can be *purely* functional only by copying entirely every relation one wishes to "change", and this is prohibitively expensive. So we need non-functional **update** operations, which have the side-effect of changing a relation in place.

SQL offers three separate commands for this, one for each of insert, change, and delete operations, with the insert command unfortunately differing from the other two by unfortunately working with only one tuple. It is cleaner to make them all relational and to avoid the ambiguity of the word, "update" by using it only to refer to relations, not tuples. It is also cleaner to isolate the update operations from the conditions determining which parts of the relation are affected. The selection-projection and joins already discussed are ample to provide this control.

Here are the three cases of the **update** statement.

update R **add** S;
update R **delete** S;
update R **using** S **change** <statements>;

S is always a relation, and the **using** S clause (which is optional) in the change command uses the natural join of S with R to select the parts of R that will change. The <statements> in this case are usually assignment statements changing values of attributes. They may contain domain algebra expressions on the right hand side. S in the **change** case may also be preceeded by a join operator, if simple **natjoin** is not enough for the task. Of course, S may be any relational expression in all three cases.

We have introduced four binary operators of the relational algebra. We have reviewed the programming notation we will use in the paper for these operators and for the unary selection-projection combination, and we have justified the programming notation as a replacement for the SQL query language for flexibility and generality.

One more unary operator is useful. Because the relational algebra is high-level, and abstracts over looping, it has no concept of an individual tuple. This is good because it requires the programmer to avoid tuple-by-tuple thinking, which can result in exceedingly poor utilization of secondary storage, as well as diminishing the level of abstraction. (SQL uses the "cursor" concept to violate this abstraction.) However, the notion of a singleton relation, i.e., containing only one tuple, is sometimes useful semantics, and we have an operator which guarantees its result to be either singleton or empty. **Pick** followed by a relational expression returns a relation consisting of a single tuple of the operand, selected nondeterministically (and not removed from the operand). (**Pick** is also the mechanism for introducing nondeterminism into the language.)

We need only now return to the actualization of virtual attributes, which we have illustrated in the context of projection. *Anywhere* in the relational algebra that an actual attribute can occur, so can a virtual attribute. It can be actualized in a join as well as in a projection; it can be tested in a selection or in a join. Furthermore, anonymous domain algebra *expressions* can often replace the name of an actual attribute, with the exception being any usage that would place an anonymous attribute in the result relation. (Even this exception is partially lifted when we come to nested relations in section 4.)

2.2 Multidimensional Databases

To illustrate the domain algebra and its interaction with the relational algebra, we will build a datacube in three dimensions. We do not have room in this paper for sophisticated algorithms which compute only partial datacubes based on space and speed tradeoffs, so we content ourselves with simply constructing the whole datacube.

We choose data which we use later in the paper to build a decision tree, from the classic paper [9].

Training	(*Outlook*	*Temperature*	*Humidity*	*Windy*	*Class*)
1	sunny	hot	high	f	N
2	sunny	hot	high	t	N
3	overcast	hot	high	f	P
4	rain	mild	high	f	P
5	rain	cool	normal	f	P
6	rain	cool	normal	t	N
7	overcast	cool	normal	t	P
8	sunny	mild	high	f	N
9	sunny	cool	normal	f	P
10	rain	mild	normal	f	P
11	sunny	mild	normal	t	P
12	overcast	mild	high	t	P
13	overcast	hot	normal	f	P
14	rain	mild	high	t	N

(The first column is a tuple identifier for later reference, and is not part of the data.) The first four attributes are the data supplied for classification, and *Class* indicates whether the tuple is a positive or a negative instance of the classification sought.

We first take advantage of the fact that *Temperature* will turn out to have no effect on the final decision tree, and use a relational algebra projection to eliminate it. Since we will be building a datacube with counts of the numbers of `N` and the numbers of `P` entries, we create these counts with the domain algebra, and we use the projection to actualize the counts. The result is a new form of *Training*, which we shall use from now on.

> **let** *N* **be equiv** + **of**
>   **if** *Class*="N" **then** 1 **else** 0
>   **by** *Outlook, Humidity, Windy*;
> **let** *P* **be equiv** + **of**
>   **if** *Class*="P" **then** 1 **else** 0
>   **by** *Outlook, Humidity, Windy*;
> *Training* <−
>   [*Outlook, Humidity, Windy, N, P*]
>   **in** *Training*;

Training	(*Outlook*	*Humidity*	*Windy*	*N*	*P*)
1,8	sunny	high	f	2	0
2	sunny	high	t	1	0
9	sunny	normal	f	0	1
11	sunny	normal	t	0	1
3	overcast	high	f	0	1
12	overcast	high	t	0	1
13	overcast	normal	f	0	1
7	overcast	normal	t	0	1
4	rain	high	f	0	1
14	rain	high	t	1	0
5,10	rain	normal	f	0	2
6	rain	normal	t	1	0

(Since tuple order does not matter in relations, but can help the reader, we have rearranged the tuples for clarity.) We see that two tuples have in two cases contributed to a negative or a positive count, and that, since both are negative or both are positive, *Temperature* has no effect on the final classification. This means that the problem is reduced to three dimensions and so is easy to visualize. (The number of dimensions makes no difference to the calculations.) We also see that *Training* contains a Cartesian product of the values for *Outlook, Humidity,* and *Windy*, and so is not sparse: in such a case, the full datacube may be constructed with minimum relative overhead of space.

Now we build the datacube. This requires three steps. The first sums the counts in the *Windy* direction. The second sums the counts, including the sums for *Windy* from the first step, in the *Outlook* direction. The third step sums all counts, including previous sums, in the *Humidity* direction. (The order in which these directions is chosen is irrelevant; any one of 3!=6 loops will give the same result. In d dimensions, there will be d steps, and d! ways of ordering them.) Figure 1 may be helpful in visualizing the process and the result.

The implementation involves equivalence reduction and renaming in the domain algebra, and projection and update in the relational algebra. The first step is

> **let** *N* **be** *totN*;
> **let** *P* **be** *totP*;
> **let** *Windy* **be** "ANY";
> **let** *totN* **be equiv** + **of** *N* **by**
>   *Outlook, Humidity*;
> **let** *totP* **be equiv** + **of** *P* **by**
>   *Outlook, Humidity*;
> **update** *Training* **add**
>   [*Outlook, Humidity, Windy, N, P*] **in**
>   [*Outlook, Humidity, totN, totP*] **in**
>   *Training*;

Note that summing over *Windy* means grouping by the complementary attributes, *Outlook* and *Humidity*, and that the projection that actualizes these sums also retains *Outlook* and *Humidity.*

The tuples that are added to *Training* by this first step are

Training	(*Outlook*	*Humidity*	*Windy*	*N*	*P*)
1,2,8	sunny	high	ANY	3	0
9,11	sunny	normal	ANY	0	2
3,12	overcast	high	ANY	0	2
7,13	overcast	normal	ANY	0	2
4,14	rain	high	ANY	1	1
5,6,10	rain	normal	ANY	1	2

These form the front face (*Outlook, Humidity*) of figure 1.

The second and third steps are identical to the first, but with the attribute names replaced as follows.

Step	Sum Attribute	Group Attributes
1	*Windy*	*Outlook, Humidity*
2	*Outlook*	*Humidity, Windy*
3	*Humidity*	*Outlook, Windy*

The second step adds the six tuples corresponding to the (*Humidity, Windy*) face of figure 1; and the third step generates the twelve tuples of the (*Outlook, Windy*) face.

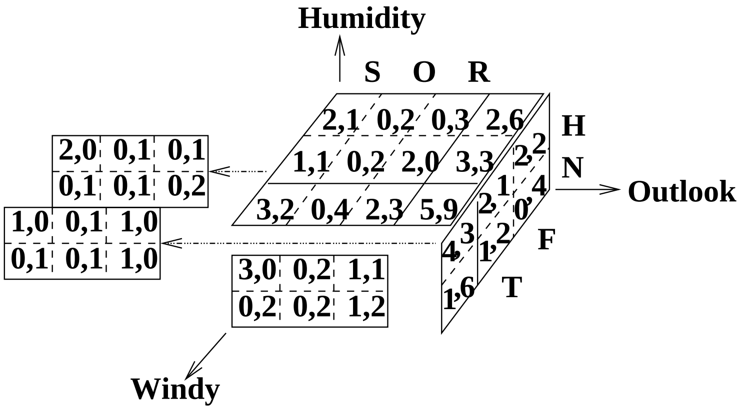

Figure 1: The DataCube for the Weather Classification

3 Attribute Metadata

None of the foregoing discussion is new with this paper, although it may come as a surprise to some readers that the classical relational algebra, minimally extended, and augmented by an independent domain algebra, can get so far with analytical programming and multidimensional database applications. (It can also be used for topological analysis, logic programming, spatial data and G.I.S., temporal data, transitive closure, inference engines, and inheritance and instantiation, among other topics well beyond the scope of this paper.) In this section, we propose a new construct.

We saw in section 2.2 that a loop of d steps was needed to build a d-dimensional datacube, and that what changed from step to step was the set of attributes involved. To write that code generally as a loop, and for any number of attributes, we allow a type **attribute** for attributes, and we introduce, temporarily (it will be replaced later by the more general **transpose** operator), an operator, **AttribsOf**, which creates a relation of all the attributes of the operand. Finally, let us introduce syntax that permits relational expressions, whose result is a relation on a single attribute of type **attribute**, to be used anywhere an unordered set of attributes was formerly allowed, i.e., in projection lists and after the **by** clause of an equivalence reduction.

As for any metadata, we need two special operators, **eval** and **quote**. **Eval** applies to any metadata variable (which must be a singleton, unary relation on an attribute of type **attribute**) and replaces the variable by its value. We will see an example below. **Quote** applies to any attribute name, and converts it to **attribute** metadata. Section 3.2 illustrates this.

3.1 The General DataCube

Here is the example of section 2.2 implemented as a single loop, using these ideas. Note that we can now use any operator of the relational algebra on sets of attributes.

```
o       let N be totN;
o       let P be totP;
n       domain attr attribute;
n       relation AllAttribs(attr) <-
          AttribsOf Training;
n       //Outlook, Humidity, Windy, N, P
n       relation ClassAttribs(attr) <-
          {(N), (P)};
n       relation TotAttribs(attr) <-
          {(totN), (totP)};
n       PropAttribs <-
          AllAttribs diff ClassAttribs;
n       LoopAttribs <- PropAttribs;
n       while [] in LoopAttribs
n       { Attrib <- pick LoopAttribs;
n// Pick the next of Outlook, Humidity, Windy
n         update LoopAttribs delete Attrib;
n// and don't use it again
o         let eval Attrib be "ANY";
o         let totN be equiv + of N
          by (PropAttribs diff Attrib);
o         let totP be equiv + of P
          by (PropAttribs diff Attrib);
o         update Training add [AllAttribs] in
          [PropAttribs diff Attrib
            union TotAttribs]
          in Training;
n       }
```

In the above program, the lines prefixed “o” are the old code from section 2.2, adapted for the general loop. The lines prefixed “n” are new code needed to initialize and use the set of attributes that control the loop. Inspection of the differences from section 2.2, and the discussions in sections 2 and 2.1, will make the meaning clear.

3.2 The Decision Tree

With attribute metadata, we can take the first step in building a decision tree, both for the specific three-dimensional example of weather classification, and in general. This involves building a datacube in which the aggregates are information values. We will be looking for minimum information values to choose an attribute as a node of the decision tree (because information is a measure of surprise, and, once we have the decision tree it should be able to predict exactly (no surprise) the class of any test tuple). This means that we must connect an attribute to the minimum information value, and that means we need attribute metadata.

The information datacube is built in $2d-1$ steps in d dimensions (instead of the d steps needed for the ordinary datacube we just built). Figure 2 shows why two extra steps are needed in the three dimensional case: the small cube schematically shows the aggregates generated by the first three steps, and the aggregates remaining to be generated by steps 4 and 5. The generalization to any number of dimensions is easy.

The numbers on the big cube are the information values computed by the code below. If we think of the cube as made up of $4\times3\times3 = 36$ blocks, the subsequent minimization must find the smallest number on any block with more than one number, and identify the attribute that is normal to this face. (For example, the minimum of 0.892, 0.788, and 0.694, on the corner block, is the latter, and the face it is on is normal to the *Outlook* attribute. This will make *Outlook* the root of the decision tree.)

As for the first datacube, we will approach the implementation in two stages of generalization. In the first stage, we show how to write each step of the loop explicitly for the specific three-dimensional example. In the second stage, we use attribute metadata to write the whole loop generally. Here are five lines of preliminary domain algebra, followed by step 1 explicitly.

```
let N be totN;
let P be totP;
let np be N + P;
let npinp be np×lg(np) −N × lg(N) −
   P × lg(P);
let inf be totinf;
//1. Outlook, Humidity
let totN be equiv + of N
   by Outlook, Humidity;
let totP be equiv + of P
   by Outlook, Humidity;
let totinf be (equiv + of npinp
   by Outlook, Humidity)/
   equiv + of np
   by Outlook, Humidity;
let accum be quote Windy;
let Windy be "ANY";
update Training add
   [Outlook, Humidity, Windy,
     N, P, inf, accum] in
   [Outlook, Humidity,
     totN, totP, totinf] in
   Training;
```

Steps 2 and 3 look very like this, apart from the same permutation of attributes that we saw for the datacube. These three steps generate 24 of the 33 aggregates shown in figure 2. Step 4 returns to *Outlook–Humidity* to complete the six aggregates indicated in figure 2, and step 5 revisits *Humidity–Windy* to find the remaining three. Here is step 4.

```
//4. Outlook, Humidity
let totN be equiv + of
   if Windy="ANY" then 0 else N
   by Outlook, Humidity;
let totP be equiv + of
   if Windy="ANY" then 0 else P
   by Outlook, Humidity;
let totinf be (equiv + of
   if Windy="ANY" then 0 else npinp
   by Outlook, Humidity)/
   equiv + of
   if Windy="ANY" then 0 else np
   by Outlook, Humidity;
let accum be quote Windy;
let Windy be "ANY";
update Training add
   [Outlook, Humidity, Windy,
     N, P, inf, accum] in
   [Outlook, Humidity,
     totN, totP, totinf] in
   Training;
```

Note that we avoid accumulating over the already computed aggregates corresponding to "ANY" values. Apart from this, the code is identical to step 1.

The generalization, using metadata, to relations on any set of attributes, is easily achieved along the same lines as above, and we mention only that

```
let accum be quote Windy;
```

and its variants can be simply replaced by

```
let accum be Attrib;
```

The only significant change is that the sequence of attributes that were randomly **pick**ed in the first d steps must be repeated in the same order in the final $d-1$ steps. This requires storing them, together with sequence numbers, in a temporary relation *Loop2Attribs*, and rearranging the control statements of the loop so *Loop2Attribs* can be initialized. These changes illustrate new idioms but no new language ideas, so we omit the code.

The next, and last, task in building the decision tree is to find the mimimum information values in the right groupings and then to search through the datacube we have just generated to select the tuples that make up the decision tree. For this, we need the **transpose** operator, which in turn needs nested relations.

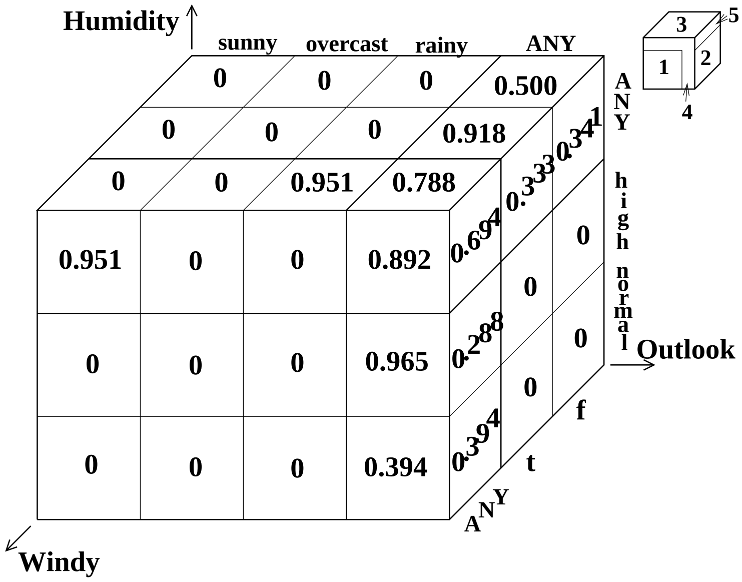

Figure 2: The DataCube for the Weather Decision Tree

4 Nested Relations

With the domain algebra and a cleaned-up relational algebra, we get nested relations for free (syntactically, that is: the implementation is more difficult, although everything we are about to say can be built with non-nested relations). The linguistic step is to subsume the relational algebra into the domain algebra.

Nested relations are relations whose attribute values may themselves be relations. Thus, to work with them, we need a formalism to create new relation-valued attributes from existing ones. The domain algebra, with relational operations incorporated into it, is the required formalism.

Although nested relations add nothing whatsoever to the *functionality* of the "flat" relational and domain algebras, which is what we have discussed so far, they do at times simplify our *thinking*, and so have a rightful place in secondary storage programming. They also repair an aesthetic inegality of the data types in the programming language: numbers, strings, and other scalar data types have hitherto had privileges that relations have not, namely the ability to be included as values in relational tuples. Nesting gives relations these privileges, too.

To review the ideas, consider the relation *ShoppingBaskets*(*xact, item*) which was mentioned in section 2.1 and which we will be using for association mining in section 4.1. Here is the domain algebra, with a new, relation-making operator, to convert the *item* attributes to a nested relation.

let *xactset* **be relation**(*xact*);

If this were actualized, each of the original tuples would have a singleton *xactset*(*xact*) relation, which is not very interesting. Let us use the relational **union** operator, now legal to be used in the domain algebra, to combine all items for each transaction.

let *xacts* **be equiv union of** *xactset*
by *item*.

Actualizing this latter by the projection

SBsets <− [*item, xacts*] **in**
ShoppingBaskets;

gives one tuple for each item, containing the item, *item*, and the set of associated transactions, *xacts*(*xact*).

We see from this the importance of being able to name the relational operators.

Similarly, relational operators can be used in scalar expressions of the domain algebra, to combine nested relations horizontally within each tuple.

To complement the **relation** construct in the domain algebra, which groups a set of attributes into a nested relation, we need a mechanism to flatten a nested relation by removing a level of nesting. Just as the nesting mechanism, **relation**, creates singleton nested relations, the inverse must start with a singleton nested relation, or else the resulting values will not fit into the tuple. Besides the nondeterministic **pick**, a projection onto the attribute resulting from a **red**uction aggregation is a good means of producing singleton results. This is a good way to unnest, because we can make such a result anonymous. Then, only the name of the nested relation is known, not the name of the attribute in it, and we can make the convention that, in such a case, the level of the attribute is raised one notch to become an attribute in the containing relation. Thus, no new syntax is needed for unnesting.

Here is an example, continuing the above, in which we count the number of items in each *xact* set, and raise the count to be an attribute of the containing relation that results.

  let *count* **be** [**red** + **of** 1] **in** *xacts*;

Since *count* is a virtual attribute created by a projection, it would normally be a nested relation, with the attributes named in the projection list. However, the projection list contains a domain algebra expression, not a name, and so *count* has no attribute. Because the domain algebra expression is a reduction aggregate, it has only one value, and so *count* can now be a flattened attribute containing only this value (in the parent tuple). Actualization could take the form

  SBsetsAgg <− [*item, xacts, count*]
    in *SBsets*;

Here is a very small collection of sample data for each of the above relations. In the three nested relations, a line separates each tuple, so we can see the singletons and how they merge into a single tuple.

ShoppingBasket

(*item*	*xact*)
`milk`	2
`bread`	1
`bread`	2
`bread`	3

⇒

(*item*	*xactset*) (*xact*)
`milk`	2
`bread`	1
`bread`	2
`bread`	3

⇓

SBsets

(*item*	*xacts*) (*xact*)
`milk`	2
`bread`	1
	2
	3

⇒

SBsetsAgg

(*item*	*xacts* (*xact*)	*count*)
`milk`	2	1
`bread`	1	3
	2	
	3	

The research literature on nested relations has focussed on algebras at the relational level, and got rather stuck on the unpleasant property of the nest and unnest operators at this level that they are not inverses of each other. But the interesting aspect of nesting is the useful manipulations. Since these, and unnesting, are entirely subsumed in the domain algebra with no new syntax or operators required, we think we have a useful formalism without awkward problems.

The following section illustrates the value of nesting and shows the above constructs, before we return in section 5.1 to the weather classification problem.

4.1 Association Data Mining

Association data mining [8, 1] attempts to find rules of the form "if one set of items occurs in a given situation, then a second set of items also occurs", for specific sets of items. The usual motivation offered is to help retail stores to discover associations among sets of products in customers' shopping baskets. *ShoppingBaskets* gives an example which appeared early in the literature.

ShoppingBaskets (*xact*	*item*)	*SingletonRules'* (*item'*	*item*)	*cover/ confden*
8	`beans`	`beans`	`rice`	1/2
9	`beans`	`beer`	`bread`	1/2
2	`beer`	`beer`	`butter`	1/2
5	`beer`	`beer`	`milk`	1/2
1	`bread`	`bread`	`beer`	1/5
2	`bread`	`bread`	`butter`	4/5
3	`bread`	`bread`	`coffee`	3/5
4	`bread`	`bread`	`milk`	2/5
7	`bread`	`butter`	`beer`	1/5
1	`butter`	`butter`	`bread`	4/5
2	`butter`	`butter`	`coffee`	3/5
3	`butter`	`butter`	`milk`	2/5
4	`butter`	`coffee`	`bread`	3/3
6	`butter`	`coffee`	`butter`	3/3
1	`coffee`	`coffee`	`milk`	1/3
3	`coffee`	`milk`	`beer`	1/2
4	`coffee`	`milk`	`bread`	2/2
2	`milk`	`milk`	`butter`	2/2
4	`milk`	`milk`	`coffee`	1/2
9	`rice`	`rice`	`beans`	1/2
10	`rice`			

The second relation, *SingletonRules'*, lists all possible associations of singleton *antecedent* sets (*item'*) and singleton *consequent* sets (*item*). The virtual attributes, *cover* and *confden*, are counts of the number of transactions associated, respectively, with *item* and *item'*, and with *item'*. These two quantities are subject to user-specified criteria for selecting rules: *confden*, which is the "cover" of the antecedents, must exceed some absolute number, say 3 in this case (or a fraction of the total number of transactions, say 0.3); *cover* is the cover of the rule, i.e., of the intersection of the set of transactions associated with the antecedent and the set of transactions associated with the consequent; the ratio *cover/confden*, called the "confidence" of the rule, must exceed a magnitude specified by the user, say 0.8.

Here is the domain and relational algebra to generate singleton rules under these two criteria.

  let *cover* **be equiv** + **of** 1 **by** *item*;
  let *confden* **be equiv** + **of** 1 **by** *item*;
  let *item'* **be** *item*;
  SingletonRules <− [*item'*, *item*] **where**
    item' $\neq$ *item* **and** *cover/confden* $\geq$ 0.8
    in *ShoppingBaskets* **natjoin**
    [*xact, item', confden*]
    where *confden* $\geq$ 3 **in** *ShoppingBaskets*;

SingletonRules', shown above, would be produced by the last line with both criteria set to 0; *SingletonRules* can be thought of as the selection on *SingletonRules'* that gives the rules

  if {`bread`} *then* {`butter`}
  if {`butter`} *then* {`bread`}

if {`coffee`} *then* {`bread`}
if {`coffee`} *then* {`butter`}

(Note that, although the code defining *cover* and *confden* is identical, different values result because actualization is done in different contexts.)

So far we have not used nested relations. Nor have we produced more than singleton rules. The interesting problem is to deal with all possible *sets* of items. We do this in two stages. First, we find all possible item sets and their associated transaction sets and covers. Then we use a nested adaptation of the above code to discover the rules. The code we show for this procedure is not optimally efficient; publication space precludes greater sophistication.

Finding all the item sets over the *cover* threshhold, and their associated transaction sets, starts with nesting operations like the one we considered in section 4.

let *items* **be relation**(*item*);
let *xactset* **be relation**(*xact*);
let *xacts* **be equiv union of** *xactset*;
let *cover* **be** [**red** + **of** 1] **in** *xacts*;
SBsets <− [*items, xacts*] **where**
  cover $\geq$ *mincover* **in** *ShoppingBaskets*;

This gives singleton sets of items, and the associated transaction sets. To extend the result to all sets of items, we use a form of transitive closure. This needs six statements of the domain algebra, which we show as the first six statements in the next paragraph, and a recursive view which we do not have room here to explain. The result is the relation *SBsetsClos*(*items, xacts*), with three tuples from *SBsets*, such as ({`bread`}, {`1,2,3,4,7`}), and four new tuples, such as ({`bread, butter, coffee`}, {`1, 3, 4`}).

The second part of the calculation adapts the SingletonRules code to nested relations.

let *items'* **be** *items*;
let *xacts'* **be** *xacts*;
let *items"* **be** *items* **union** *items'*;
let *xacts"* **be** *xacts* **natjoin** *xacts'*;
let *items* **be** *items"*;
let *xacts* **be** *xacts"*;
let *confden* **be** [**red** + **of** 1] **in** *xacts*;
let *cover* **be** [**red** + **of** 1] **in** *xacts"*;
GeneralRules <− [*items'*, *items*]
  where (**not**(*items* **comp** *items'*)
  and [] **in** *xacts"*
  and *cover*/*confden* $\geq$ `0.8`
  in (*SBsetsClos* **natjoin**
  [*xacts'*, *items'*, *confden*]
  where *confden* $\geq$ `3` **in** *SBsetsClos*);

What this code does is

1. prepare to rename the nested relations *items* and *xacts*, and to rename the results back again,
2. define unions of *items* sets (nested relations) and intersections of *xacts* sets,
3. count the numbers of transactions for the antecedent set of *items* and for the whole rule (intersecting *xacts* sets for each item), and
4. join *SBsetsClos* with itself, as above for singleton rules, selecting on the two criteria.

The result is seven rules, including the four singleton rules we found earlier, and

if {`coffee`} *then* {`bread, butter`}
if {`bread, coffee`} *then* {`butter`}
if {`butter, coffee`} *then* {`bread`}

This calculation is trivially extended to deal with multiple-attribute properties instead of the unary nested relation, *items*(*item*).

5 The Transpose Operator

Now that we have described relational nesting, and given it a good workout by implementing associative data mining, we can return to attribute metadata and classification mining. The **transpose** operator creates a nested relation which includes an attribute of type **attribute**. It effectively converts each tuple of an ordinary relation into a set of attribute-value pairs, which we can use to write code which examines both attributes and values, as needed to build the decision tree.

The result of **transpose** is a binary nested relation, with one attribute of type **attribute** and the other capable of holding any type of data, which we call type **universal**. Consider the statements

domain *attr* **attribute**;
domain *val* **universal**;
let *xpose* **be** [*attr, val*] **transpose** A, B, C;

and the relation

$R(A$	B	$C)$	*xpose* (*attr*	*val*)
`"a"`	`1`	`true`	`A`	`"a"`
			`B`	`1`
			`C`	`true`
`"b"`	`1`	`false`	`A`	`"b"`
			`B`	`1`
			`C`	`false`

We see the (virtual) result of **transpose** on all the attributes of R.

We can now show how to find all attributes of a relation (the temporary **AttribsOf** operator in section 3.1).

AllAttribs <− [**red union of**
  [*attr*] **transpose**
  Outlook, Humidity, Windy, N, P]
  in *Training*;

Transpose can be used to transpose not only all the attributes of a relation, but also any subset of them.

5.1 Classification Data Mining

With nested relations and the **transpose** operator, we can now take the final step in building the decision tree for the weather classification problem, and in general. Our goal is to construct the decision tree represented as a set of rules, as in the relation DT.

DT

(*Outlook*	*Humidity*	*Windy*	*pivot* (*accum*)	*notANY* (*attr*)	*NP*) (*attr*)
`ANY`	`ANY`	`ANY`	`Outlook`		*N* *P*
`sunny`	`ANY`	`ANY`	`Outlook` `Humidity`	`Outlook`	*N* *P*
`overcast`	`ANY`	`ANY`	`Outlook` `Humidity`	`Outlook`	*P*
`rainy`	`ANY`	`ANY`	`Outlook` `Windy`	`Outlook`	*N* *P*

`sunny`	`high`	`ANY`	`Outlook`
			`Humidity`
			`Windy`
				`Outlook`
				`Humidity`
					N

`sunny`	`normal`	`ANY`	`Outlook`
			`Humidity`
			`Windy`
				`Outlook`
				`Humidity`
					P

`rainy`	`ANY`	`f`	`Outlook`
			`Humidity`
			`Windy`
				`Outlook`
				`Windy`
					P

`rainy`	`ANY`	`t`	`Outlook`
			`Humidity`
			`Windy`
				`Outlook`
				`Windy`
					N

The new attributes, *pivot, notANY*, and *NP*, are defined

let *pivot* **be pick equiv union of**
relation(accum) **by** *Outlook, Humidity, Windy*;
domain *attr* **attribute**;
domain *val* **universal**;
let *notANY* **be** [*attr*] **where** *val*≠`"ANY"`
in [*attr, val*] **transpose** *Outlook, Humidity, Windy*;
let *NP* **be** [*attr*] **where** *val*≠`0` **in**
[*attr, val*] **transpose** *N*, *P*;

The relation *DT* can be used as a set of rules by selecting singleton *NP* values (either *N* or *P*), as in

if Outlook = {`"overcast"`} *then* `P`

or

if Outlook = {`"sunny"`} *and*
Humidity = {`"high"`} *then* `N`

etc. Or it can be construed as a tree whose root is the tuple where *notANY* is empty, first-level nodes where |*notANY*| = 1, and so on. Searching this tree closely follows the method for extracting it from the original *Training* set, which we now give.

We first find the minimum information, and the attribute responsible for it, for each block in figure 2.

let *ct* **be** [**red** + **of** 1] **in** *notANY*;
Min <−
[*Outlook, Humidity, Windy, ct, notANY, NP, pivot*]
in [*Outlook, Humidity, Windy, accum*]
where *inf* = **equiv min of** *inf*
by *Outlook, Humidity, Windy* **in** *Training*

We can then write a loop of *d* steps to find each of the *d* levels of the decision tree. Here we give the first and the second steps; the second becomes the general step just by changing the value for *ct* in the selection.

// Step 0.
DT0 <−
[*Outlook, Humidity, Windy, notANY, NP, pivot*]
where *ct* = 0 **in** *Min*;
// Step 1.
let *pivot′* **be** *pivot*;
let *pivot″* **be** *pivot* **union** *pivot′*;
let *pivot* **be** *pivot″*;
let *ctNP* **be** [**red** + **of** 1] **in** *NP*;
DT1 <−
[*Outlook, Humidity, Windy, notANY, NP, pivot*]
in [*Outlook, Humidity, Windy, notANY, NP, pivot″*]
in ((**where** *ct* = 1 **in** *Min*)
[*notANY* **natjoin** *pivot′*]
[*pivot′*]**where** *ctNP* > 1 **in** *DT0*)

When *ctNP* = 1, we have a leaf node, or a final rule, as indicated above, and so that branch of the tree-building may stop. The whole loop must stop when all branches have ended in leaves.

Note that the natural join in this code joins two relations on equality of join attributes which are themselves relations.

DT is just the **union** of all the *DT*i's generated by the loop.

Postscript Two other classifications, One-rule and Bayesian, use much simpler calculations than the decision tree, and can be built as simple variants of the above, using only the top-level aggregates.

6 Conclusion

We have implemented two forms of data mining, classification by decision tree, and association, using only the relational and domain algebras. However, it is not the purpose of this paper to contribute to data mining, which is why we have presented only simple and inefficient algorithms.

The purpose of the paper has been to demonstrate that the relational algebra, with suitable notation, together with the domain algebra (and relational recursion and looping), suffice to do sophisticated programming. We have used nested relations, which come effectively for free with the domain algebra. We have introduced in this paper metadata of type **attribute**, and simple related operators and syntax, to write the general loops needed for datacube and decision tree construction. We expect these new constructs to be generally useful in relational database programming.

The attribute metadata type has recently been implemented. Every other technique used in this paper (domain algebra, nested relations, etc.) is implemented and running in one or more research systems.

7 Acknowledgements

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. This work was motivated by work on *spatial* OLAP and data mining, supported by the Networks of Centres of Excellence program through the GEOIDE Project, GEODEM. Andrey Rozenberg has built attribute metadata and its operators into the *relix* implementation of the language.

This paper is published in the Lecture Notes in Computer Science series by Springer-Verlag (http://www.springer.de/comp/lncs/index.html), who hold the copyright on non-electronic reproduction.

References

[1] R. Agarwal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM International Conference on Management of Data, May 26-28, 1993*, pages 207–16, Washington, D.C., May 1993. ACM Press.

[2] E.F. Codd, S.B. Codd, and C.T. Salley. Providing OLAP to user-analysts: An IT mandate. Technical report, E. F. Codd & Associates, Hyperion Solutions, Sunnyvale, CA, 1993. http://www.arborsoft.com/essbase/wht_ppr/coddps.zip, http://www.arborsoft.com/essbase/wht_ppr/coddTOC.html.

[3] P. Fischer and S. Thomas. Operators for non-first-normal-form relations. In *Proc. 7th COMPSAC*, pages 464–75, Chicago, November 1983.

[4] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichert, M. Venkatarao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997.

[5] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. ACM Symposium on Principles of Database Systems*, pages 124–38, March 1982.

[6] A. Makinouchi. A consideration on normal form of not-necessarily normalized relations in the relational model. In A. G. Merten, editor, *Proc. 3rd Internat. Conf. on Very Large Data Bases*, pages 447–53, October 1977. examples of nest, recursive nest; discusses normalization, dep.

[7] T. H. Merrett. Experience with the domain algebra. In C. Beeri, U. Dayal, and J. W. Schmidt, editors, *Proc. 3rd Internat. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 335–46, San Mateo, California, July 1988. Morgan Kaufmann Publishers Inc.

[8] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro, editor, *Knowledge Discovery in Databases*, pages 229–48. AAAI/MIT Press, 1991.

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