BASICS: Domain Algebra

Relational Information Systems — Chapter 3.1

(Revised 99/10)

November 30, 1999

Copyright ©1999 Timothy Howard Merrett
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation in a prominent place. Copyright for components of this work owned by others than T. H. Merrett must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to republish from: T. H. Merrett, School of Computer Science, McGill University, fax 514 398 3883.

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

Alongside, and almost independently of, the relational algebra is an algebra on attributes. This permits us to fill the obvious gap in the relational algebra that includes its inability to do, for instance, arithmetic.

This gap is partly filled in ad hoc ways by many commercial database systems, but we seek a deeper solution. We invoke both the *principle of abstraction* and the *principle of closure* (see Chapter 2.1, on the Relational Algebra). The principle of closure says that operations on attributes must give attributes. The principle of abstraction says that the context of the attribute(s) is unimportant. So the algebra on attributes will avoid all mention of specific relations.

This abstraction of attributes away from relations is the only subtlety in what follows, which is otherwise straightforward. It is an important subtlety, because it permits us to separate relational operations from operations on attributes. This in turn leads to important intellectual simplification, because we can think independently of each of two sides of any problem.

The algebra on attributes is called the *domain algebra*, partly because of the alliterative difficulty of "attribute algebra", and partly to emphasize its treatment of attributes independently of relations.

The domain algebra has two main components (see the figure, A Taxonomy of Aldat, in Chapter 2.1): scalar operations, and aggregation operations. In the context of the table representation of relations, these can be thought of, respectively, as "horizontal" and "vertical" operations, working, respectively, horizontally within the tuples, and vertically among the tuples.

1 Scalar Operations

The scalar operations support arithmetic, logic, and string processing on the typed values of the attributes of a relation. For any data type which is allowed as an attribute, its allowed operations can be supported by scalar operators. But scalar operators are confined to work only within each tuple independently, hence the appellation "horizontal".

An example involving simple arithmetic suffices to give the idea for any operation on any attribute types.

let *Total* **be** *Asst* + *Exam*;

Although we might have in mind evaluating this sum given a relation such as

NewMk

(*Student*	*Asst*	*Exam*)
Smith	25.	60.
Jones	28.	66.
Brown	20.	50.
Hung	24.	58.
Raman	24.	66.,

the domain algebra is independent of this or any particular relation.

It is nonetheless convenient to *think* of the result in the context of, say, *NewMk*, as long as we note that the sum is *not* an attribute of this relation:

NewMk

(*Student*	*Asst*	*Exam*)	*Total*
Smith	25.	60.	85.
Jones	28.	66.	94.
Brown	20.	50.	70.
Hung	24.	58.	82.
Raman	24.	66.	90.

Total is a *virtual* attribute at this stage. It is not *actual*ized in any relation. So we must ask how we can get it evaluated. Note that it appears outside the parenthesized set of attributes of *NewMk*, not in *NewMk*.

To actualize a virtual attribute, we use the relational algebra. The most obvious way is through a projection, but, in general, anywhere in the relational algebra an actual attribute may be used, a virtual attribute may appear, too. Here is a pair of statements to define and actualize *Total* in this way.

let *Total* **be** *Asst* + *Exam*;
Result <− [*Student, Total*] **in** *NewMk*;

Here is the result.

Result

(*Student*	*Total*)
Smith	85.
Jones	94.
Brown	70.
Hung	82.
Raman	90.

Although the operations are trivial, the consequences of the principle of abstraction are not always obvious. Since they underly the operations we will look at next, we belabour the point just a little bit more.

Frequently Asked Questions

- What if *Asst* and *Exam* are also in some other relation?
  Nothing is affected: Total could be actualized there, too.
- What if *Asst* and *Exam* each come from different relations?
  Use relational algebra to put the relations together before actualizing.
- What if we update *NewMk* after actualizing *Total*?
  Total is an actual attribute of Result and unaffected by changes to other relations.

A couple of special cases of these horizontal operators are useful. Here is how we can define a *constant* attribute, i.e., one with the same value in every tuple.

let *One* **be** 1;

And we can *rename* an attribute.

let *Final* **be** *Exam*;

We actualize these in *NewMk*

Course <− [*Student, Asst, Final, Total, One*] **in** *NewMk*;

Course

(*Student*	*Asst*	*Final*	*Total*	*One*)
Smith	25.	60.	85.	1
Jones	28.	66.	94.	1
Brown	20.	50.	70.	1
Hung	24.	58.	82.	1
Raman	24.	66.	90.	1

2 Aggregation Operations

While the scalar operations work horizontally (in the table form of a relation), within each tuple, the aggregation operations of the domain algebra work vertically, down the columns. This permits, for example, summing all the values of an attribute.

There are two classes of aggregation operation, *reduction* and *functional mapping*. They each subdivide into two subclasses.

2.1 Reduction

Totalling is the obvious example of an aggregation operation. In the domain algebra

let *TotMk* **be red** + **of** *Total*;

sums all the values of the attribute named, *Total*.

We can aggregate in lots of other ways.

let *MaxMk* **be red max of** *TotMk*;
let *MinMk* **be red min of** *TotMk*;
let *ProdMk* **be red** × **of** *TotMk*;

find the maximum and mimimum marks, and, if we should want it, the product of all marks.

Each of these aggregate virtual attributes is a constant attribute, in that, in any relation in which it is actualized, it will have the same value for all tuples. As we did above, for concreteness, we might think of each in the context of the source relation, for instance, *Course*.

Course

(*Student*	*Asst*	*Final*	*Total*	*One*)	*TotMk*	*MaxMk*	*MinMk*
Smith	25.	60.	85.	1	421	94	70
Jones	28.	66.	94.	1	421	94	70
Brown	20.	50.	70.	1	421	94	70
Hung	24.	58.	82.	1	421	94	70
Raman	24.	66.	90.	1	421	94	70

But any programmer who actualized them in this context would invite complaints of waste.

Class <− [*TotMk, MaxMk, MinMk*] **in** *Course*;

Class

(*TotMk*	*MaxMk*	*MinMk*)
84.2	94	70

This gives us three of the five aggregate operators usually boasted by comercial database systems. The other two are *count* and *average*.

let *Count* **be red** + **of** *One*;
let *AvgMk* **be** *TotMk*/*Count*;

Note that in the last we have combined two aggregate virtual attributes with a scalar operation, division. This uses the principle of abstraction, which is what permits us to think of virtual attributes as values associated with each tuple of any relation they can be evaluated in. Because attributes are closed under the domain algebra (principle of closure), we can build expressions.

let *AvgMk* **be** (**red** + **of** *TotMk*)/(**red** + **of** 1);

Let us replace *Class*.

Class <− [*TotMk, AvgMk, MaxMk, MinMk*] **in** *Course*;

Class

(*TotMk*	*AvgMk*	*MaxMk*	*MinMk*)
84.2	84.2	94	70

In addition to the conventional aggregations (which are hardwired into the usual database system, and not derived from a single concept and syntactic construct as here), we can calculate product (above) and two Boolean aggregates corresponding to universal and existential quantification. The following examples allow us to say that every mark is no more than the maximum and some mark is the same as the maximum.

let *ALLleMax* **be red and of** *Total*≤*MaxMk*;
let *SOMEeqMax* **be red or of** *Total*=*MaxMk*;

Both these Boolean aggregates will have the value **true** if actualized from a relation such as *NewMk*.

These are six basic operators that may be used by the reduction aggregation: +, ×, **min**, **max**, **and**, **or**. Operators that are not associative, such as string concatenation, **cat**, or not

commutative, or both, such as $-$, $/$, or **mod** may not be used, because tuple order does not matter in relations, but different orders would produce different results. (Two more Boolean aggregate operators are $=$ and $\neq$. What do they mean?)

As well as the basic operators, such as $+$, **min**, and **max**, and constructs based on them for counting and averaging, which give the conventional aggregations, we may make arbitrary constructs. For example, the variance is

let *VarMk* **be** (**red** + **of** (*Total*−*AvgMk*)↑2)/*Count*↑2;

The term "reduction" originates from the similar operator of APL. Since this is *A P*rogramming *L*anguage for arrays, whose elements are ordered by index value, APL reduction supports the noncommutative and nonassociatve operators.

Equivalence reduction allows reduction to be applied to groups of tuples within a relation. These groups are *equivalence classes* induced by a specified attribute or set of attributes having the same value within the group. For an example, we can continue in the vein of course marks, but this time for several courses.

let *STot* **be equiv** + **of** *Mark* **by** *S#*;
let *CTot* **be equiv** + **of** *Mark* **by** *C#*;

StuCour

(*S#*	*C#*	*Mark*)	*STot*	*CTot*
1	1	73.	233.	213.
1	2	82.	233.	147.
1	3	78.	233.	78.
2	1	64.	64.	213.
3	1	76.	141.	213.
3	2	65.	141.	147.

Once again, we show the virtual attributes in the context of a source relation, but sensible actualizations would look more like

ByStu <− [*S#, STot*] **in** *StuCour*
ByCou <− [*C#, CTot*] **in** *StuCour*

Note that the capabilities are symmetrical, even though the example relation is presented above in an order which favours the visualization of *STot*.

With the machinery that we now have, we can combine domain and relational algebras to do interesting things. Here are three matrices

$$A\begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 1 \\ 3 & 0 & 1 \end{pmatrix} \quad B\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad AB\begin{pmatrix} 3 & 5 & 2 \\ 3 & 2 & 1 \\ 4 & 5 & 1 \end{pmatrix}$$

in which AB is the matrix product of A and B. A relational representation of all three matrices could use triples, with element values and two indices, omitting tuples for elements with value 0.

A(*ValA*	*I*	*J*)	*B*(*ValB*	*J*	*K*)	*AB*(*ValAB*	*I*	*K*)
1	1	1	1	1	1	3	1	1
2	1	3	1	1	2	5	1	2
2	2	2	1	2	1	2	1	3
1	2	3	1	3	1	3	2	1
3	3	1	2	3	2	2	2	2
1	3	3	1	3	3	1	2	3
						4	3	1
						5	3	2
						1	3	3

The task for the relational and domain algebras is to compute the relation AB given relations A and B. The nice consequence of the principle of abstraction is that we can divide the problem into two parts and solve them separately. The product we wish to compute is

$$AB_{i,k} = \sum_j A_{i,j} B_{j,k}.$$

First, we must combine A and B before we can do any calculations. Our reflex should be to use the natural join, and reflection confirms that this is right: in each row, i, of A, the jth element must be linked to the jth element of each column, k, of B. We thus have a relation on attributes *ValA*, I, J, K, and *ValB* to work with.

Second, we think about the arithmetic. We must multiply the two values for the common J, then we must sum over all values of J for a fixed I and K. In the domain algebra, we do not need to specify that J ranges, but only that I and K are fixed: we need an equivalence reduction by I and K.

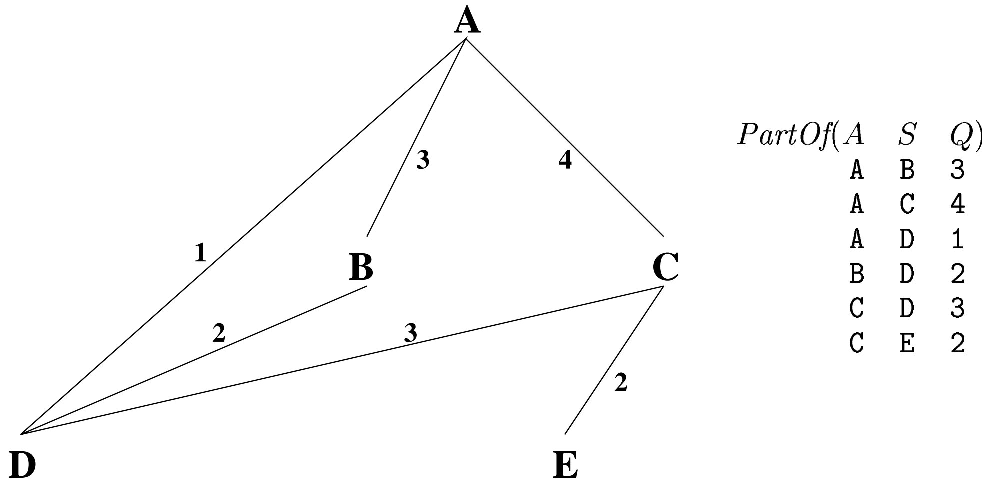
let *ValAB* **be equiv** + **of** *ValA* × *ValB* **by** I, K;
AB <− [*ValAB, I, K*] **in** (*A* **ijoin** *B*);

This code is brief, and, with a fairly unsophisticated implementation, is an efficient way to multiply two sparse matrices (lots of 0s) which may even be too big to fit into RAM.

Here are intermediate steps, shown after the join has been calculated, with *ValAB* shown only for $(I, K) = (1, 1)$ and $(I, K) = (3, 1)$, for clarity.

A **ijoin** *B*

(*ValA*	*I*	*J*	*K*	*ValB*)	*ValA*× *ValB*	*ValAB*
1	1	1	1	1	1	3
1	1	1	2	1	1	:
3	3	1	1	1	3	4
3	3	1	2	1	3	:
2	2	2	1	1	2	
2	1	3	1	1	2	3
2	1	3	2	2	4	:
2	1	3	3	1	2	
1	2	3	1	1	1	
1	2	3	2	2	2	
1	2	3	3	1	1	
1	3	3	1	1	1	4
1	3	3	2	2	2	:
1	3	3	3	1	1	



PartOf(*A*	*S*	*Q*)
A	B	3
A	C	4
A	D	1
B	D	2
C	D	3
C	E	2

Figure 1: A Bill-of-Materials

A related calculation is more elaborate: the "explosion" of a bill of materials into its full transitive closure plus quantities. A *bill of materials* is the structure that shows the components of an artefact, such as a product assembled from various pieces which themselves are subassemblies. Figure 1 is a reprise of the bill-of-materials shown in section 4 of chapter 1.1, on relations, together with a relational representation. (The term "explosion" refers to the image that is frequently drawn of the components all separated from each other, but aligned, with lines connecting the points where they are joined. These lines often terminate in drawings of the bolts and nuts used to join the components, which look as though they have been blown out of the assembly.)

Considering domain algebra first, the calculation used in the explosion process must multiply the quantities on edges that lie in the same path, and sum these products for paths that terminate in the same (sub)assemblies. Thus, the 3 `B`s that make up an `A` each have 2 `D`s. The 4 `C`s that make up an `A` each have 3 `D`s. And there is 1 `D` directly involved in an `A`. So, in total $3 \times 2 + 4 \times 3 + 1 = 19$ `D`s directly or indirectly make up an `A`.

This is essentially just the domain algebra that went into multiplying two matrices. We will return to consider some small variations needed.

The relational algebra is essentially the transitive closure used in the *Ancestor* calculation in section 3.4 (recursion) of chapter 2.1 (relational algebra). However, we cannot use the natural composition to link edges in the same path because of the possibility that two pairs of edges connect the same two points (such as `A` and `D`) and the further possibility that the products of the quantities may be the same (such as 3×2 and 2×3, if the 4 on edge `AC` had been changed to a 2). To avoid the automatic duplicate elimination, we must use the natural join instead, to retain the linking nodes (`B` and `C` in the above example) that differentiate the tuples. Because of this, and because of the fact that the natural join of a relation with itself (*PartOf*) on different attributes (S on one hand and A on the other) leads to ambiguous attribute names, the natural join must be accompanied by renaming. This can be done by domain algebra.

let A' **be** A; **let** S' **be** S; **let** Q' **be** Q;
let Q'' **be equiv** + **of** $Q \times Q'$ **by** A, S';
let Q''' **be** $Q + Q''$; **let** Q **be** Q''';
Explo **is** $[A, S, Q]$ **in** $[A, S, Q''']$ **in** (*PartOf* $[A, S$ **ujoin** $A, S']$

Sales			
(*President*	*Year*	*Amount*)	*Cum*
Smith	1994	150	150
Smith	1995	175	325
Smith	1996	200	525
Brown	1996	200	525
Brown	1997	210	735
Brown	1998	225	960

Figure 2: Functional Mapping

$[A, S', Q'']$ **in** (*Explo* [*S* **ijoin** *A'*] $[A', S', Q']$ **in** *PartOf*));

In the code, the attributes of *PartOf* are renamed (by adding primes) before joining with the recursively defined *Explo*. Q'' gives the sum of products (the matrix multiplication). Q''' adds to this any values previously generated for shorter paths between the same two nodes. Finally, Q is renamed from Q''' in order that the attributes of *Explo* are the same as those of *PartOf* so the recursion can continue. In the domain algebra there seems to be a circular definition here, Q in terms of Q''', in terms of Q and Q'', in terms of Q and Q', in terms of Q. But the interplay with the relational algebra is such that the projections always disambiguate and avoid the circularity.

Despite this interplay, one is easily able to think independently about the domain algebra and the relational algebra sides of the problem, once familiar with the two algebras and how they are put together.

2.2 Functional Mapping

Although reductions are restricted to associative and commutative operators because of the independence of relations from tuple order, non-commutative and non-associative operators are useful. We could use them if an ordering were *induced* on the tuples of a relation by the value of one of the attributes, or of a set of attributes. We could then also obtain *cumulative* totals, and similar results for other associative and commutative operators.

Figure 2 shows a relation which combines annual sales totals with information on who was president of the company at the time. In this context, we can think about cumulative sales for the company,

let *Cum* **be fun** + **of** *Amount* **order** *Year*;

and the figure shows *Cum* as a virtual attribute. Note that there is a functional dependence, *Year* → *Amount*, and that a new functional dependence, *Year* → *Cum*, is generated. Because the *Amount* is the same for the two tuples for `1996`, *Cum* must also not change within this year. (It is to illustrate this point that the apparently irrelevant attribute, *President*, is shown.)

This family of aggregations is called *functional mapping* because it is a function from one function (*Year* → *Amount*) to another (*Year* → *Cum*). Such functions of functions are usually called *functionals*. Common mathematical examples of functionals include integration (for instance, $g(y) = \int_0^y f(x)\mathrm{d}x$ maps f to g) and differentiation. Functional mapping can be used to give a form of numerical integration, which is very crude because the data points are predetermined by the tuples of the relation and cannot be selected for optimality by

(*Num*	*Seq*)	*AltS*	*AltP*
3	1	3	3
12	2	9	4
20	3	11	5
60	4	49	12
180	5	180	15

Figure 3: Alternating Sum and Product

the integration process. Appropriate applications might be time-series forecasting and data smoothing, which are simplified forms of numerical integration.

For order-dependent operators (either non-commuative or non-associative) such as $-$, we must specify what "cumulative" means. It is best to use $-$ to produce an alternating sum, $/$ to give an alternating product, and so on. Thus, the value of the virtual attribute in tuple r_i for the ith distinct value of the ordering attribute, x (in ascending order), for operator β is

$$r_i[\textbf{fun}\ \beta\ \textbf{of}\ y\ \textbf{order}\ x] = r_i[y]\ \beta\ (...\ \beta\ (r_3[y]\ \beta\ (r_2[y]\ \beta\ r_1[y]))...)$$

where y is the operand attribute and **fun** β **of** y **order** x is the resulting virtual attribute.

Equivalently, we can specify this in a way which is closer to an implementation.

$$accum = \textbf{if}\ i = 1\ \textbf{then}\ r_1\ \textbf{else}\ r_i[y]\ \beta\ accum$$

where *accum* is the accumulator, which gives the value of $r_i[\textbf{fun}\ \beta\ \textbf{of}\ y\ \textbf{order}\ x]$ for each tuple, r_i, $i = 1, ..., N$.

Figure 3 shows an alternating sum and an alternating product ordered by *Seq*.

let *AltS* **be fun** $-$ **of** *Num* **order** *Seq*;
let *AltP* **be fun** $/$ **of** *Num* **order** *Seq*;

We can see that the values of *AltS* are the sequence, $3, 12-3, 20-12+3, 60-20+12-3$, and $180-60+20-12+3$. Similarly, *AltP* takes on the values $3, 12/3, 20/12 \times 3, 60/20 \times 12/3$, and $180/60 \times 20/12 \times 3$.

The usual alternating sum would be $3, 3-12, 3-12+20, 3-12+20-60$, and $3-12+20-60-180$, but it is easy to convert to this from the result of the functional mapping. Other definitions of the behaviour of order-dependent operators would not give an alternating sum at all, for instance

$$accum = \textbf{if}\ i = 1\ \textbf{then}\ r_1\ \textbf{else}\ accum\ \beta\ r_i[y].$$

is the same as $r_1[y] - \sum_{i=2}^{N} r_i[y]$.

Useful new operators to introduce in the context of functional mapping are **pred** and **succ**. These are defined to give the *cyclic* predecessor and successor, respectively, of the value of the operand attribute. (The above definition using an accumulator is overruled for **pred** and **succ**. **Pred** and **succ** have no useful scalar meanings.) The reason for cyclicity is that it is sometimes useful (for example, in geometrical calculations), and that it is easy to write code to go from a cyclic result to a noncyclic answer, but not the other way around. Figure 4 shows the successor of each word in a text,

let *Next* **be fun succ of** *Word* **order** *Seq*;

Partial functional mapping adds a grouping facility to functional mapping in the same way that equivalence reduction does for reduction. Figure 5 shows the example for

Text

(*Word*	*Seq*)	*Next*
Algebraic	1	data
data	2	processing
processing	3	techniques
techniques	4	can
can	5	enable
enable	6	applications
applications	7	programmers
programmers	8	to
to	9	work
work	10	with
with	11	units
units	12	of
of	13	data
data	14	larger
larger	15	than
than	16	a
a	17	single
single	18	computer
computer	19	word
word	20	Algebraic

Figure 4: Word Sucssession

DivSales

(*Div*	*Year*	*Amount*)	*DCum*
A	1997	80	80
A	1998	110	190
B	1997	60	60
B	1998	75	135
C	1997	90	90
C	1998	110	200

Figure 5: Partial Functional Mapping

let *DCum* **be par** + **of** *Amount* **order** *Year* **by** *Div*;

Sale *Amount*s are accumulated over the years for each *Div*ision.

Partial functional mapping, not surprisingly, is related to partial integration.

A useful idiom allows us to count the number of *different values* an attribute has in a relation. One way is to project the relation on the attribute of interest, then do a **red** + **of** 1 to count the tuples. It would be useful to get this result without doing the projection, especially if the count is to be used for further comparisons or calculations within the relation. Here is the domain algebra that will do this, for the example of figure 2.

let *CountPres* **be red max of fun** + **of** 1 **order** *President*;

This counts the number of different *President*s (2) appearing in the relation. First, we accumulate a count for each different president, then we find the maximum of these accumulating counts. Here are the results

Sales (*President*	*Year*	*Amount*)	**fun** + **of** 1	**red max of** ..
Smith	1994	150	2	2
Smith	1995	175	2	2
Smith	1996	200	2	2
Brown	1996	200	1	2
Brown	1997	210	1	2
Brown	1998	225	1	2

A similar idiom can find the number of different values of some attribute in groups of tuples determined by another attribute. For example, in the context of figure 5, we could count the number of different years reported by each division (2 each: the following idiom will give this result even if the divisions report quarterly and another attribute told us the quarter).

let *CountYear* **be equiv max of par** + **of** 1 **order** *Year* **by** *Div* **by** *Div*;

Note the two '**by** *Div*''s are needed. One could put parentheses from (**par** to the first **by** *Div*) for clarity.

3 Nested Relations

Several of the relational algebra facilities we have examined so far add little new functionality. For example, σ-joins may be simulated by a combination of domain algebra and natural join. QT-selectors can mostly be implemented by equivalence reduction and projection. But we have introduced them separately because they fit into a framework and are useful. They embody in syntax different ways of thinking about problems. This is valuable because different people think in different ways.

The present section introduces a further conceptual extension which also adds no new functionality. It is a new way of thinking inherent in previous ideas, and requires almost no new syntax.

Nested relations repudiate the fourth characteristic of relations presented in section 1 of chapter 1.1, on relations. This was the requirement that *the value in each row under a given column is "simple"*. This requirement was vague as to what "simple" means, but in practice it boiled down to excluding relations as values of attributes. This is arbitrary, except that it stems from implementation difficulties. Now that we have enough "flat" relational and

domain algebra in our arsenal, we can see how the restriction can be lifted and the result yet implemented using flat operations.

The term "nested" relations comes from our willingness, now, to allow attributes in relations to have values in each tuple that are themselves relations. A motivation for this liberation comes from programming languages and the ideal that all data types should be "first-class". So far, relations have not been first class: they cannot be used anywhere strings, for instance, are used, specifically within relations. If we allow relations to be attributes, we promote them, and we permit new ways of thinking about relations containing relations containing relations, etc.

Because the simplicity requirement enforced by "first normal form" on our relations hitherto, nested relations are also known as $\neg$1nf or nf^2 (non-first-normal-form) relations.

Apart from some minor new syntax for declarations, we need no syntactic changes but only an insight. This insight is that we *subsume the relational algebra into the domain algebra.* If attributes may now be relations, the language must support operations on relational attributes. The domain algebra supports attribute operations, and the relational algebra operates on relations. So all we must do is permit relational operations in the domain algebra.

We now work through a sequence of examples to show the possibilities, starting with two-level relations (one level of nesting). Here is a declaration (from [JS82]), showing the only new syntax we will need.

```
domain Authors(authors);
domain Descriptors(descriptors);
relation Books(Authors, title, price, Descriptors) <- {
    ({("A1"),("A2")},"T1","P1",{("D1"),("D2")}),
    ({("A2")},"T2","P2",{("D1"),("D2")}),
    ({("A1")},"T3","P1",{("D1"),("D2"),("D3")})
};
```

It is useful to think of *Books* as a relation of three structured tuples, as shown below.

Books (*Authors* (*authors*)	*title*	*price*	*Descriptors* (*descriptors*))
A1 A2	T1	P1	D1 D2
A2	T2	P2	D1 D2
A1	T3	P1	D1 D2 D3

This is reminiscent of the structured relations we rejected in chapter 1.1 as adding no new information. Note, however, that the flat version of this relation would need nine tuples instead of three. Four of these would expand the first tuple, because every combination in the Cartesian product of {(A1),(A2)} with {(D1),(D2)} must appear. (Any other choice would add new information, for instance, that D2 is not used to describe A1.)

Now we pose a query on *Books*.

Find all titles whose authors contain A2 and whose descriptors contain D1 and D2.

```
relation A2(authors) <- {("A2")};
relation D1D2(descriptors) <- {("D1"),("D2")};
```

A2D1D2 <− [*title*] **where** *Authors* **sup** *A2* **and** *Descriptors* **sup** *D1D2* **in** *Books*;

First we create two relations with the query data, then we compare the relation-valued attributes to them in a T-selector. Note that the comparisons are σ-joins that produce nullary results, and hence are Booleans. It is these two σ-joins that are now done in the domain algebra. We could spell this out

let *Auth1* **be** *Authors* **sup** *A2*;
let *Des12* **be** *Descriptors* **sup** *D1D2*;
A2D1D2 <− [*title*] **where** *Auth1* **and** *Des12* **in** *Books*;

but it is not necessary to use **let** and invent temporary names.

So far we have shown how we can nest relations and pose queries on them with no new syntax or concepts. But we have not shown that it could all be done with flat relations. Here is a representation of the nested relation, *Books*.

Books(*Authors*	*title*	*price*	*Descriptors*)
0	T1	P1	0
1	T2	P2	1
2	T3	P1	2

.Authors

(*.id*	*authors*)
0	A1
0	A2
1	A2
2	A1

.Descriptors

(*.id*	*descriptors*)
0	D1
0	D2
1	D1
1	D2
2	D1
2	D2
2	D3

We label each *Authors* set by an identifier, and each *Descriptors* set similarly. In fact, these identifiers can just be tuple numbers for *Books*, as shown here; but usually they are independently generated surrogates. The *Authors* attribute is typed so that its values are represented by the identifiers, and the same with *Descriptors*. Then we create one new relation for each of these two attributes, containing all three sets of values, discriminated by a new *.id* attribute.

To formulate the same query as above, we translate the **and** to **ijoin** and the **where** to **ijoin** or to **icomp**.

A2D1D2 <− [*title*] **in** *Books* [*authors* **icomp** *.id*]
((*.Authors* **sup** *A2*) **ijoin** (*.Descriptors* **sup** *D1D2*));

The implementation of nested relations follows just these two steps.

1. For each nested attribute, create a single relation containing its attributes, plus a set-identifying attribute, and replace each nested relation in each tuple by the surrogate value linking to this new attribute.

2. For each query, translate relational operations expressed in the domain algebra to top-level relational operations on these new relations. Because the new relations represent all the relations for the attribute, operations in the relational and domain algebras may need to add grouping attributes. (Generally, reductions and functional mappings translate respectively to equivalence reductions and partial functional mappings, for example.)

In the example above, the nested relations are simple sets. Here is an example (from [DL87]), again with two levels, but with full relations.

Employee						
(*ENo*	*name*	*Children*			*Training*	)
		(*name*	*date*	*sex*)	(*CNo*	*date*)
105	John	Jane	800510	F	314	791010
		Eric	821005	M	606	810505
					714	820620
123	Anne	Maria	751112	F	315	810613
					423	820711
153	Bruce				314	791010
205	Ian	Bob	701016	M	314	791010
		Steve	750115	M		

Let us

find employees without children.

NoKids <– [*ENo*] **where not** ([ ] **in** *Children*) **in** *Employee*;

Here we have used the nullary projection, [ ] **in** *Children* ("something in *Children*"), as a domain expression to give the Boolean value that is used in the top-level T-selector.

Find names of children of employees who took course 314.

let *C* **be** [*name*] **in** *Children*;
CN314 <– [*C*] **where** ([ ] **where** *CNo*="314" **in** *Training*) **in** *Employee*;

We must note that the result is itself a nested relation, *CN314*(*C*(*name*)). It has the value

CN314
(*C*)
(*name*)
Jane
Eric
Bob
Steve

3.1 "Nest" and "Unnest" Operators.

It seems plausible that the intention of the query is to obtain the simple set of names, {Jane, Eric, Bob, Steve}, in which case we must remove the nested structure of the result so far. This is called *unnesting*, and can be achieved in two steps.

First, the three relations in the three tuples must be reduced to one, the single desired set. This is easily done, within our new framework of subsumption of the relational algebra in the domain algebra, by reduction.

red ujoin of [*name*] **in** *Children*

Note that **ujoin** is a commutative, associative operator, and so is legal in reduction.

Projecting the **red ujoin** still gives a nested relation, but a singleton. So the second step is to remove a level of nesting. This we can do unambiguously in a singleton relation. (In a relation of more than one tuple, containing a nested subrelation, removing a level would

require the system to make a decision about how to combine the different tuples; but the programmer should do this explicitly.)

The way we lift a level is through *anonymity*, i.e., simply not giving it a name. *CN314*, above, contains a nested relation named *C* because we created *C* in a **let** statement. We can avoid this by just writing the expression in the projection list, instead of in a **let** statement.

Here is the new code to give a flat result, *CN314*.

> *CN314* <– [**red ujoin of** [*name*] **in** *Children*]
>   **where** ([ ] **where** *CNo*="314" **in** *Training*) **in** *Employee*;

Since we find ourselves doing a lot of projections to create unary relations while processing nested relations, and since there is a frequently used syntax for this sort of operation, we introduce "syntactic sugar" as a shorthand.

> *Children.name* ≡ [*name*] **in** *Children*]

So, finally,

> *CN314* <– [**red ujoin of** *Children.name*]
>   **where** ([ ] **where** *CNo*="314" **in** *Training*) **in** *Employee*;

This code will unnest a relation with several nested attributes, but the resulting single sets must be combined into flat tuples by Cartesian products. A sequence of **ijoins**s of the results of the **red ujoin**s will do the job, with renaming if the nested relations have common attributes.

For completeness, if we can unnest from a nested relation to a flat relation, we should be able to *nest* a flat relation into a nested one. We also do this in two steps.

The first step is to add a level by giving a name to the new relation attribute. We have already shown **domain** declarations to structure an attribute leading to initializing a new nested relation. We can use the same mechanism to give a collective name to a set of attributes, but it reads better if we couch it in the domain algebra.

> **let** *Training'* **be relation** (*CN0, Date*);

would define a nested attribute for, possibly, a flat relation, *Employee*(*ENo, CNo, Date*).

This first step does not tell us how to group the tuples of the flat relation into tuples of the nested result. In fact, each instance of *Training'*, in each tuple of *Employee*, will be a singlton relation. For this, **equiv ujoin** is needed.

> **let** *Training* **be equiv ujoin of** *Training'* **by** *ENo*;

would combine the singleton *Training'*s into a multi-tuple relation for each different *ENo*.

We show the nesting of *EmployeeFlat* in figure 6 to give the above nested relation, *Employee*.

> **let** *Training* **be equiv ujoin of relation** (*CNo, date*) **by** *ENo*;
> **let** *name* **be** *Cname*;
> **let** *date* **be** *Cdate*;
> **let** *Children* **be** [*name, date, sex*] **in equiv ujoin of relation** (*Cname, Cdate, sex*)
>   **by** *ENo*;
> *Employee* <– [*ENo, name, Children, Training*] **in** *EmployeeFlat*;

In the above code, *ENo* is singled out from the other attributes for the group by, for obvious reasons. We can write more general code than this if we include in the syntax a way to express "all attributes in the relation except ..." Here is code to do the same job without

EmployeeFlat						
(*ENo*	*name*	*Cname*	*Cdate*	*sex*	*CNo*	*date*)
105	John	Jane	800510	F	314	791010
105	John	Jane	821005	M	606	810505
105	John	Jane	821005	M	714	820620
105	John	Eric	800510	F	314	791010
105	John	Eric	821005	M	606	810505
105	John	Eric	821005	M	714	820620
123	Anne	Maria	751112	F	315	810613
123	Anne	Maria	751112	F	423	820711
153	Bruce	$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$	314	791010
205	Ian	Bob	701016	M	314	791010
205	Ian	Steve	750115	M	314	791010

Figure 6: An Unnested Version of *Employee*

referring to any attributes except the ones involved in the nesting.

let *Training* **be equiv ujoin of relation** (*CNo, date*) **by** !{*CNo, date*};
let *name* **be** *Cname*;
let *date* **be** *Cdate*;
let *Children* **be** [*name, date, sex*] **in equiv ujoin of relation** (*Cname, Cdate, sex*)
    by !{*Cname, Cdate, sex*};
Employee <– [!{*Cname, Cdate, sex, CNo, date*}, *Children, Training*] **in**
    [!{*CNo, date*}, *Training*] **in** *EmployeeFlat*;

The first line, above, gives the same *Training* relation for each combination of `John, Jane` and `John, Eric`, and the rightmost projection in the last line replaces the attributes *CNo, date* by this new relation on these attributes. The fourth line and the second projection give the *Children* relation including `Jane` and `Eric` for `John`, and replace *Cname, Cdate, sex* by *Children*(*name, date, sex*).

Unnesting is likely to be more useful than nesting. There is an further asymmetry in that *nest* does not always invert *unnest*. A simple example is an attempt to nest the final form we had for *CN314*: this could only re-introduce the extra level, *C*(*name*), but the three tuples in the original *CN314* could never be reconstructed. More elaborate examples have additional attributes to distingush the tuples, but two or more tuples have the same value for this attribute in the nested relation.

On the other hand, *unnest* is always the inverse of *nest*.

Because these two transformations behave in this difficult way, and are not very important to boot, we do not define special operators. It is more useful to add the simple syntax of the **relation** grouping and the semantics of level lifting through anonymity. With the general subsumption of the relational algebra into the domain algebra, these give us nesting and unnesting if we want them. We also get a general method for formulating queries, which are much more interesting.

3.2 Deeper Nesting

Without more ado, we can move on to deeper nesting. Here is a nested relation with four levels.

Faculty							
(*dept*	*Profs*						)
	(*name*	*sal*	*office*	*Students*			)
				(*name*	*office*	*Courses*	)
						(*numb*	*grade*)
CS	Pat	55	312	Kim	300	612	A
						617	A-
				James	300	612	B+
						617	A-
	Sue	42	322	Chris	328	506	A
	Luc	48	319	Kim	300	612	A
						617	A-
EE	Pat	55	312	Kim	300	612	A
						617	A-
				James	300	612	B+
						617	A-
	Denis	52	412	Joe	425	530	B+
				Peter	426	531	A-

It is a little strange, in that data is repeated, but this helps us make some points in the following query.

Find depts with two A *grades.*

Note that this query crosses from the lowest level (*grade*) to the highest (*dept*). Here is the formulation.

let *CA* **be red** + **of if** *grade* = "A" **then** 1 **else** 0;
let *SCA* **be** [*name*, [*CA*] **in** *Courses*] **in** *Students*;
let *PSCA* **be** [**red ujoin of** *SCA*] **in** *Profs*;
let *ct* **be red** + **of** *CA*;
Ans <– [*dept*] **where** ([*ct*] **in** *PSCA*) $\geq$ 2 **in** *Faculty*;

Figure 7 follows the workings step by step. In this example, for legibility, most virtual attributes are shown as they would be actualized by intermediate projections. Also, the eight relations *PSCA* (one for each of the four tuples of *Profs* for *Dept*=CS, and one for each of the four tuples of *Profs* for *Dept*=EE) are shown as two, so that the redundancies of the three repetitions of each different value are not shown.

Note the three lifts via anonymity. The first is for *CA* when it is actualized in *SCA*. The second and third are for *PSCA* and *ct* when they are used in the *where* clause of the T-selector on *Faculty*. This eliminates the three intermediate levels in the query.

Throughout this formulation of the query, the rule must be kept in mind that no lifting can be done except in a singleton relation. Thus, *CA*, *PSCA* and *ct* are each defined as reductions.

Faculty (dept	Profs (name	sal	office	Students (name	office	Courses (numb	grade)	CA)	SCA (name	CA)	PSCA (name	CA)	ct)
CS	Pat	55	312	Kim	300	612	A	1	Kim	1	Kim	1	2
						617	A-	1			James	0	2
				James	300	612	B+	0	James	0	Chris	1	2
						617	A-	0					
	Sue	42	322	Chris	328	506	A	1	Chris	1			
	Luc	48	319	Kim	300	612	A	1	Kim	1			
						617	A-	1					
EE	Pat	55	312	Kim	300	612	A	1	Kim	1	Kim	1	1
						617	A-	1			James	0	1
				James	300	612	B+	0	James	0	Joe	0	1
						617	A-	0			Peter	0	1
	Denis	52	412	Joe	425	530	B+	0	Joe	0			
				Peter	426	531	A-	0	Peter	0			

Figure 7: A Nested Query Across Four Levels

3.3 Recursive Nesting

The ultimate nested relation defines a *recursive data type*, in which nesting is permitted to arbitrary depth. Here is a reformulation of the cost problem for a bill-of-materials from this point of view.

```
relation Assembly(Id, Assembly, Cost) <-{ ("A", {("A1", {("A11", dc, 6),
                                                         ("A12", dc, 5)}, 3),
                                                  ("A2", dc,             2)}, 4)

                                        };
```

Assembly is recursively defined in terms of itself. The depth of nesting is controlled by the data, with $\mathcal{DC}$ used to halt the recursion. *Cost* is interpreted at the lowest levels as the purchase cost of the raw materials. At higher levels, it is the cost of assembling the component.

The calculation of the total costs is written recursively in the domain algebra.

> **let** *CalCost* **be if** *Assembly*=$\mathcal{DC}$ **then** *Cost* **else** *Cost* + *Assembly.SumCost*;
> **let** *SumCost* **be red** + **of** *CalCost*;
> *TotalCost* <− *Assembly.SumCost*;

Here are the workings of this query in this case. Since the recursion stops with *CalCost* at the lowest levels, *CC* (*CalCost*) is the same as *C* (*Cost*) whenever *Assy* (*Assembly*) equals $\mathcal{DC}$. *SC* (*SumCost*) is the reduction of this, and its level is then lifted by the anonymity of *Assembly.SumCost* and added to *Cost* the next level up. So, working from the centre rightwards, we have $((8 + 5) + 3 + 2) + 4 = 20$.

Assembly												
(*Id*	*Assembly*									*C*)	*CC*	*SC*
	(*Id*	*Assembly*					*C*)	*CC*	*SC*			
		(*Id*	*Assy*	*C*)	*CC*	*SC*						
A	A1	A11	dc	6	6	11	3	14	16	4	20	20
		A12	dc	5	5	11						
	A2			dc			2	2	16			

References

[DL87] V. Deshpande and P.-A. Larson. An algebra for nested relations. Technical Report CS-87-65, University of Waterloo Computer Science Department, Waterloo, Ont., Dec. 1987.

[JS82] 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.