Aldat: a Retrospective on a Work in Progress

T. H. Merrett
McGill University, Montreal, Canada

Despite its immense success, the relational model of data has been underappreciated. Many wrong claims have been made to the effect that it is unable to handle complex data, to do analytical processing, or to go beyond passé, simple structured data. I have devoted most of a career in computer science to showing that relations can indeed cope with all these, without awkwardness and with minimal syntactic and conceptual extensions. Not only can relations cope; they do the job better. A further advantage of this work is integration: the same formalism that was classically used for administrative data can also be used for expert systems, for geographical information systems, for CAD-CAM, for numerical work, for data mining and for semistructured applications such as bibliographic and bioinformatic databases. Another advantage is that this integrated relational formalism is at a level of abstraction which is not only ideally suited for processing data on secondary storage but which also readily absorbs important issues in computational parallelism and in distributing data over the Internet.

I review the simple ideas needed to push the relational model to its inherent full capabilities, and show the syntactic adjustments needed to avoid the limitations of conventional and commercial implementations. The discussion is prefaced by some motivating examples, without full explanations, and terminated by a consideration of some special techniques for implementing the language constructs.

Keywords: secondary storage programming language

1. Introduction

The Aldat Project is almost complete, to use famous penultimate words. We have developed it in parallel with progress in the database research and application communities since the inception of the relational model by Codd [15,17]. Our purpose from the beginning was to produce a programming language, capable of any processing which relations could in principle support. The database community was by and large interested only in query languages, but we started our work in general programming languages for secondary storage. (It was not until half a dozen years after the relational model first appeared that work was started on database programming languages [76], leading to the current biennial workshops on "DBPL".) We also found ourselves, in the early days, having to devise data structures and algorithms to support our language constructs on secondary storage.

We followed two principles in building a general-purpose language from the relational algebra Codd originally supplied. First, add nothing which is not motivated by at least two independent applications. Second, add nothing which does not fit the given conceptual framework without distortion: existing concepts may, and should, be generalized as far as possible, and new syntax added with only the greatest reluctance. The thrust of these principles is to keep Aldat as simple as possible, and has resulted in unexpected functionality from only three or four types of relational operator, an independent "domain algebra" of three categories of operations, a fusion of the notions of procedure and relation, and some classical programming language ideas such as typing, scopes and recursion.

We have not been zealous in following any particular language paradigm to the hilt, such as declarative/functional programming, logic programming, constraint programming or object-oriented programming, but we have exploited their various strengths whenever possible and consistent. Thus, Aldat is mainly declarative (functional programming) and provides encapsulation for non-declarative code (object-oriented programming). Aldat offers recursive views (logic programming) and multi-way functions (constraint programming).

We have been doctrinaire about one consideration, namely always to work at the level of whole relations and never with individual tuples. This suits the bulk data transfers required by secondary storage, it avoids requiring a parallelizing compiler having to undo the order of loops coded by programmers, and it gives the further advantages of truly high-level languages, such as LISP and APL, that the abstractions are as deep as possible and the programmer is saved many nitty-gritty lies of code.

In Codd's relational algebra we found two tactical principles which we also observed. First, the principle of closure: operations on things (e.g., relations) must produce things of the same kind (e.g., relations). This permits arbitrary expressions to be built up from basic operations, and keeps the formalisms self-contained. Second, the principle of abstraction: operations on things (e.g., relations) should not be influenced either by the structure of the things (e.g., the tuples of data) or by their context. This is essential for modularity and for the intellectual simplification that is the goal of any programming language to deliver. We applied these principles to relations in the relational algebra and to attributes in the independent domain algebra.

With the formalization of grammars and to a large extent of semantics, and with the development of parsing and compiling techniques based on these formalizations, language development is now often considered to be a negligible achievement. Developing the concepts for a radically new language is something else. Our work has been empirical, a multi-year investigation of new applications as they arose, mainly in databases, to integrate them into a single framework satisfying all the above principles.

In the 1970s, while the first relational systems were being built and some fundamental theory investigated, we established the basic operators needed to abstract over looping and treat files of data in the same way that programming languages then treated (and still treat) numbers. This produced the domain algebra and some generalizations of relational operators into a family of quantified unary operators and two families of binary operators, one extending set-valued operations on sets and the other extending logic-valued operations on sets.

In the 1980s the database community moved on to forms of data other than the administrative data that had originally motivated database development. We established that spatial, temporal, and rule-based systems could all be built on relations and the above operations. Recursion was needed in the relational algebra, and introduced without new syntax beyond a mechanism for relational "views".

By the 1990s, the object-oriented approach was challenging the relational formalism, and we undertook to show that the two were not incompatible: if one can have an object-oriented list-processing language, one can certainly have an object-oriented relational language. Indeed, a relational language can readily subsume O-O. Because O-O is a language facility, rather than a data structure, this required us to begin fusing the fundamental ideas of programming languages with the relational ideas we had developed. Since encapsulation of state is central to O-O, and since all we need for this is a procedure mechanism [3], we had to incorporate procedures. Instead of just sticking them in, we kept to our principles and sought a common generalization. Mathematically, relations generalize functions[1], and so we had the notion of a "computation", a parametric procedural abstraction which can be invoked by the relational algebra, using different sets of input parameters under different circumstances. (In a sense, this is a reprise of an idea which goes all the way back to the very first relational implementation [80].)

Nested relations, which Codd anticipated from the outset ([15]), but which violate his "first normal form" [16], had been around ([44]) as long as DBPL, and needed attention. It is fundamental to programming languages that there should be no "second-class citizens" which are not allowed in certain contexts, but first normal form insisted that relations could not be values of attributes in the tuples of relations. Instead of being diverted

[1]Functions are the many-to-one special case of binary relations.

by the problematical nonsymmetry of "nest" and "unnest" operations [35], we looked for a formalism in which relation-valued attributes could be useful, e.g., in certain forms of data mining. Again, no new syntax was needed, just pushing the existing concepts a little further. If relations are allowed to be attributes, then the domain algebra (which operates on attributes) must subsume the relational algebra. That's all. The language can now express, for instance, basic association data mining.

For classification data mining, on the other hand, the usual implementation must loop through each of a set of attributes. This requires support of attribute "metadata": the ability to include attribute names among the data to be processed, and to convert back and forth between data and attributes. (Since nesting has united attributes and relations, attribute metadata includes relational metadata.) A nice application is building datacubes [27], fundamental to on-line analytical processing (OLAP) which the founder of the relational model said was not an intended application for relations [18]. (He didn't say it *could* not be done.)

By the present decade, work on "semistructured" data has reached fruition with the appearance of query languages which treat XML as a data structure. As with object-orientation and OLAP, it is received wisdom that relations just can't cope with the polymorphism and lack of explicit schemas that characterize semistructured data. So we looked at nested relations. They were not yet quite complete, lacking the capability of recursive nesting (which the LISP data structure has, for instance). Incorporating this made the domain algebra recursive, which was also lacking. This enables relations to deal with implicit schemas and with many of the semistructured queries. Adding suitable polymorphism to Aldat handles the rest. Previous semistructure query capabilities are a strictly special case. They can be captured by a syntactic sugar which replaces recursion by path expressions. Since the formalism for nested relations adds no new capabilities to what flat relations can accomplish (although thinking about certain problems is greatly simplified), we have achieved semistructure capabilities with flat relations.

With this recursive completion of relational potential, the following definitions are now meaningful. A *relation* is a time-varying set of tuples. A *tuple* is a mapping, also time-varying, from a set of names (*attributes*) to corresponding values. Since these values may be relations, the definitions are recursive. They also allow us to say that a *database* is a tuple. (So for that matter, is a *data structure*.) We can see that a tuple is the same thing as a *scope* in programming language terminology, and a *directory* in an operating system file hierarchy. This last observation permits us to use the same syntax for path expressions in semistructured data as a navigation tool among multiple databases stored in different directories of a single host, or, with a suitable protocol, among distributed parts of a database on several hosts.

This paper supplies technical details on the above retrospective. After some formal definitions and an outline of the main example (Section 3), we attempt to explain, in a self-contained way, the domain algebra (Section 4), the relational algebra (Section 5), relational nesting (Section 6) and the computation (Section 7). Before this, Section 2 gives some motivating examples, showing full code but without the explanations that come later. Section 8 applies these ideas to aspects of internet programming, and Section 9 discusses a couple of data structures for implementation on secondary storage which arose during the language development.

2. Motivation

This section provides some motivation for the rest of the paper by giving six Aldat examples which show the benefits of a very high-level general-purpose programming language. Matrix multiplication, inference engines, bills of materials, rotation and shear transformations, semistructured data and boolean circuits are not within the capabilities of most database languages. Here I show complete code for each of them in a language which later sections develop from the central database idea of relations. The section is not self-contained, but looks ahead to later discussions. Appendix A works through, with data, all but the last example of this section, and can be followed once the remainder of the paper has been read.

The matrix multiplication example shows that, in a sufficiently high-level language, we can avoid writing unnecessary loops. Multiplication of $n \times n$ matrices costs $\mathcal{O}(n^3)$ operations, and is written as three nested loops in low-level languages. (We can define a low-level language to be one that requires three nested loops to multiply two matrices.) Here is the product in Aldat, for matrices represented as relations, $A(i, j, a)$ and $B(j, k, b)$.

let *ab* **be equiv** + **of** *a* * *b* **by** *i*, *k*;
AB <−[*i*, *k*,*ab*] **in** (*A* **natjoin** *B*);

The second line is relational algebra, which takes the natural join of A and B on the common attribute, j, then projects the result on i, k and on a calculated attribute, ab. The first line is "domain algebra", which calculates ab as a sum of $a \times b$ over j (i.e., grouped by i and k).

The reason Aldat does not need any looping construct to multiply matrices is that all three loops are order-independent. A low-level programmer is forced to write these loops, each following a certain sequence. If the code is then to be run on parallel processors, a very clever compiler must be invoked to "parallelize" the code by undoing those sequences.

Aldat does not force the programmer into unnecessary sequencing because it is a language developed for processing data in bulk, as found on secondary storage, and its central constructs (such as natural join) abstract over looping.

The domain algebra is another central construct in Aldat, and it also abstracts over looping. The domain algebra supports an even more important abstraction, to which I shall return in the bill-of-materials example, below. This is its orthogonality to the relational algebra. Note that the domain algebra line, above, makes no reference to any relation. This greatly reduces extraneous thinking by permitting problems to be decomposed into the aspect in which attributes are processed and the aspect in which relations are processed.

Our second example is a one-line inference engine. This applies to the Horn clauses (if-then rules, where the "if" is followed by a conjunction of one or more antecedents),

Horn(*Rule#*,*Ante*,*Concl*),

and the initial facts, *Facts*(*Concl*), and uses any Horn clause whose antecedents are covered by known facts to derive new facts, which are in turn made the basis for further inferences.

NewFacts **is** *Facts* **ujoin** [*Concl*] **in**
(*NewFacts*[*Concl*: **sup** :*Ante*]*Horn*);

This is recursive code, which shows one possible mechanism for writing loops explicitly, when the order of execution does matter. The **is** construct defines a view, as opposed to the assignment, <−, above, which causes the code to be executed and the result materialized. This particular view happens to be recursive.

The **sup** operator is one of a few slight extensions to the classical relational algebra. It is related to division [17], and is a member of a family of "σ-joins" which also includes natural composition. It is **sup** which ensures that a rule is fired only if all its antecedents are known to be facts.

The outer or union join, **ujoin**, is, in Aldat, likewise a member of a family, the "μ-joins", which includes the natural join and the set operations of intersection, union, etc.

The brevity of this and the matrix multiplication code is not necessarily an end in itself. Some languages have gone overboard to the point of unreadability striving for brevity. The fact that the code is brief is rather a confirmation that the concepts of a language which was originally intended for database-like operations are both high-level and general. This one-line inference engine expands to 50 lines in a 200-line expert system shell which took a person-month to write [52]. High productivity and low error rates are also a consequence of brevity, in a profession where the number of bug-free lines of code a programmer can write in a day is essentially independent of the level of the language they are written in.

A third example provides a moderately complicated solution and illustrates the savings permitted by separating domain algebra from relational algebra. The "bill of materials" of a manufactured product describes its components and their quantities in a hierarchy of subassemblies. Figure 2 shows it for an electric wallplug.

A processing challenge is to determine total quantities needed, e.g., four screws and four connectors. This requires recursing through the hierarchy, as the inference engine did, or, more closely related, as do transitive closure of a graph or the problem of finding ancestors given parents. It also requires multiplying quantities along paths and summing these products over all paths with common endpoints, as matrix multiplication did.

The first is a relational algebra problem, the second a domain algebra calculation. Because the two are largely independent, we should be able to think of them independently, which Aldat allows. The first three lines are the domain algebra that does all the arithmetic. (The six **let** statements may be written in any order.) The last line is the recursive view that does the transitive closure and incorporates the arithmetic. We can represent the bill of materials by the relation *PartOf* (A, S, Q), with A, S, Q standing, respectively, for *assembly, subassembly* and *quantity.*

```
let $A'$ be $A$; let $S'$ be $S$; let $Q'$ be $Q$;
let $Q''$ be equiv + of $Q * Q'$ 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 :$A, S']$ $[A, S', Q'']$ in
    (Explo $[S$: natjoin :$A']$ $[A', S', Q']$ in
    PartOf));
```

I will use this example throughout the paper.

The relational and domain algebras provide a formalism for relations and their attributes much as arithmetic provides a formalism for numbers. When programming languages were developed to do arithmetic, they offered more than just the arithmetic operators. The foremost of these powerful language constructs is procedural abstraction, which parametrizes code and enables it to be written once and invoked repeatedly. Procedural abstraction has two forms, procedures and functions. Focussing on the latter, mathematics describes a function as a many-to-one mapping between domains. This is a special case of the mathematical concept of a relation, a many-to-many mapping between domains. So instead of immediately adding syntax along traditional lines to define and invoke functions, we pause to ask how to introduce a special case of the relations we already have. From a programming point of view, we can specify any of the attributes of a relation as "input", using a selection, and any (others) of its attributes as "output", using projection. Functions, by contrast, can be invoked only in one direction. We can generalize functions to a programming construct, which can be invoked in various directions with simple select and project. This construct is many-to-many but still a special case of relations because not every subset of its attributes is sufficient for an invocation. (Constraint programming languages use the term "modes" for different invocations of the same procedure using different parameters as inputs.)

We call this generalization a *computation*, which is invoked as a relation and has declaration syntax allowing **alt**ernative definitions, to respond to alternative sets of selected and projected attributes. Having this capability allows functions and their inverses to occupy a single package, and it encourages programmers to think about alternatives such as inverses.

Our fourth example is the declaration of a computation which looks at two-dimensional rotation in all possible ways. The **alt** keyword separates "**alt**-blocks" of code, one of which is selected by the implementation for execution according to whether the invocation supplies all the inputs it needs.

```
comp rotate$(x, y, x', y', \theta)$ is
{ $x' <- x * \cos\theta - y * \sin\theta$;
  $y' <- x * \sin\theta + y * \cos\theta$;
} alt
{ $x <- x' * \cos\theta + y' * \sin\theta$;
  $y <- -x' * \sin\theta + y' * \cos\theta$;
} alt
{ $x <- x' * \sec\theta + y * \tan\theta$;
  $y' <- x' * \tan\theta + y * \sec\theta$;
} alt
{ $x' <- x * \sec\theta - y' * \tan\theta$;
  $y <- -x * \tan\theta + y' * \sec\theta$;
} alt
{ $y' <- x * \csc\theta - x' * \cot\theta$;
  $y <- x * \cot\theta - x' * \csc\theta$;
```

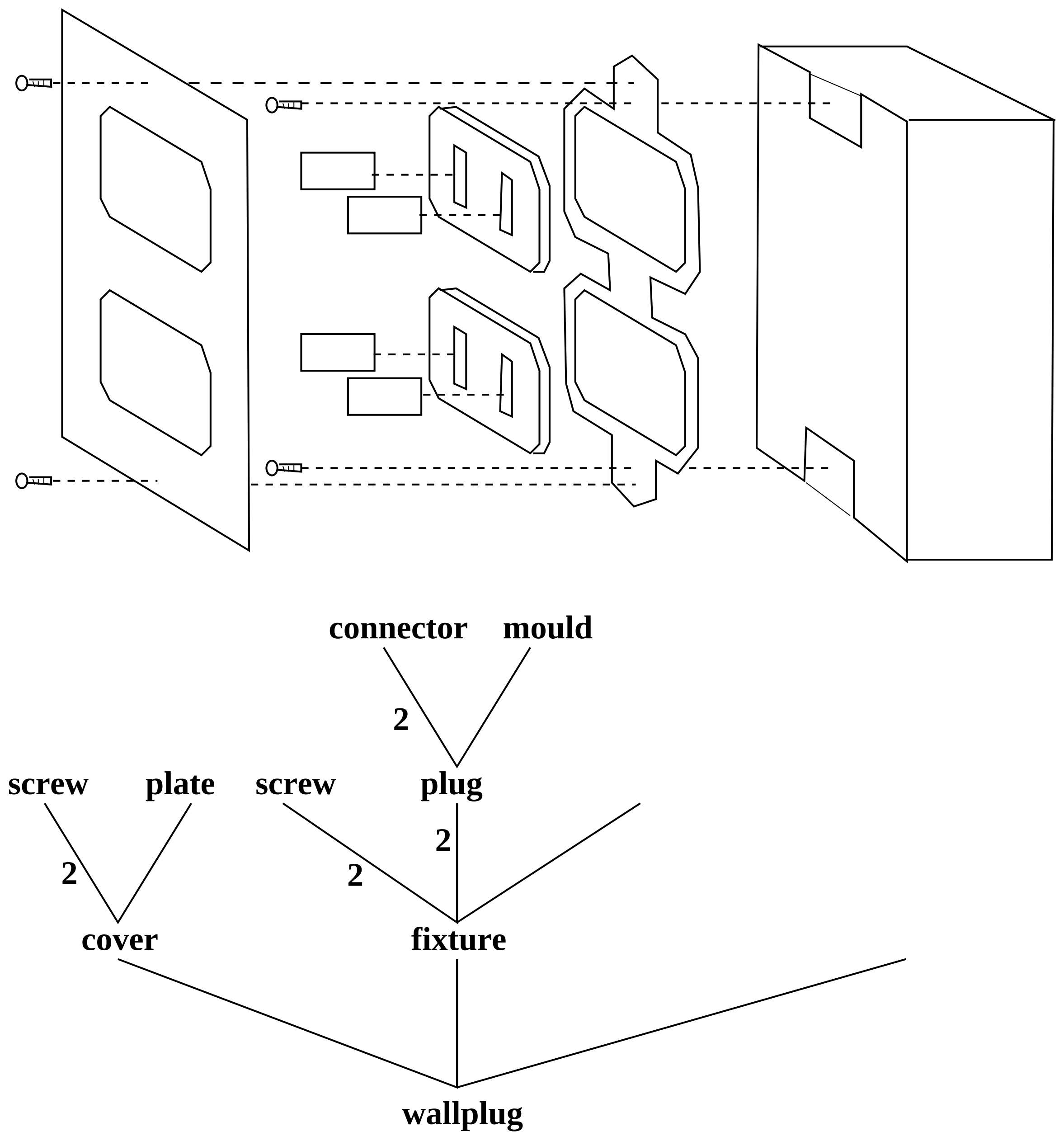

Figure 2: the wallplug bill of materials

```
} **alt**
{ $x <\!\!- y' * \csc\theta - y * \cot\theta$;
  $x' <\!\!- y' * \cot\theta - y * \csc\theta$;
} **alt**
$\theta <\!\!- \arccos((x * x' + y * y')/(x^{**}2 + y^{**}2))$;
```

We have 4 **choose** 2 = 6 possible alternatives (θ and any two of x, y, x' and y' may be given and the other two calculated), plus a seventh if the angle, θ, is the unknown.

The second pair of transformations given in *rotate* are a shear transformation and its inverse. If $\sin\theta = v/c$ they are also the Lorentz transformation of space-time and its inverse. Being encouraged to think of all possible equivalents leads to intriguing perspectives.

Aldat does not oblige the programmer to think out all possible alternatives, however: the computation could have been written with any non-empty subset of the above seven **alt**-blocks.

This **comp**utation can equally be thought of as a **comp**ressed relation: an infinite number of tuples represented by seven equivalent alternative sets of rules. From this point of view, invocation through selection-projection operations of the relational algebra is restricted to any of seven modes (for this example) of setting parameters as either input or output.

A fifth example follows immediately from approaching relations from the point of view of programming languages. Good programming languages avoid "second-class citizens". These are language elements, such as arrays, which have fewer privileges than other elements, such as integers. Arrays are second-class citizens in a language whose procedures can return integers but not arrays.

In a relational language, relations should not be second class: if a relational attribute can be a scalar, such as integer, real, boolean or string, it should also be allowed to be a relation. This leads to nested relations such as the family tree shown in Table 2.

To process such nested relations, we need to subsume the relational algebra under the domain algebra. Since attributes can be relations, the algebra that handles attributes must also be able to handle relations.

This requirement in turn forces us to use explicit, in-fixed operators for the relational algebra, particularly joins, and will make us modify the usual query-language syntax to a programming-language syntax.

This insight enables us to deal with nested relations using no new syntax.

PERSON in Table 2 not only has nested relations as attributes, but they are recursively nested. To find all the *Name*s anywhere in *PERSON* requires recursive domain algebra.

```
let Nom be Name ujoin
    [red ujoin of
        [red ujoin of Nom] in
        CHILDREN] in
    FAMILY;
[red ujoin of Nom] in PERSON
```

Since the nested relation *PERSON* could be derived from a markup of the text

> Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James and Jane had Tom in 1961 and Sue in 1962).

the ability to query it fully is a significant component of a semistructured query language. No new language has had to be defined and, in fact, nested relations and the expanded domain algebra can be implemented directly on "flat" relations of the kind that Codd classically defined. Thus, given flat relations and the relational and domain algebras, we get a major aspect of semistructured data for free.

Our objective in dealing with semistructured data is not to implement special cases of XML and semistructured query languages with nested relations and syntactic sugar, but to replace both by more general capabilities based finally on flat relations and recursive algebras.

The sixth example illustrates the incompleteness of the work, as pointed out in the title of this paper. We would like to be able to use metadata to construct boolean circuits (and many similar problems). Metadata is data which is itself the names of relations and attributes. Here is ordinary data defining the behaviour of basic logic

PERSON									
(*Name*	*FAMILY*								)
	(*Conj*	*Wed*	*CHILDREN*						)
			(*DoB*	*Name*	*FAMILY*				)
					(*Conj*	*Wed*	*CHILDREN*		)
							(*DoB*	*Name*)	
Ted	Alice	1932	1934	Mary	Alex	1954	1956	Joe	
			1935	James	Jane	1960	1961	Tom	
							1962	Sue	

Table 2: recursively nested family tree.

gates.

and(*x*	*y*	*z*)	*or*(*x*	*y*	*z*)	*not*(*x*	*y*)
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Now we use metadata to combine some gates into a circuit.

CIRCUIT			
(*ASSY*	*SUBA*	*PIN*	*LABEL*)
nor	or	x	a
nor	or	y	b
nor	or	z	c
nor	not	x	c
nor	not	y	d

The objective is to use *CIRCUIT* to calculate the new relation, $nor(x, y, z)$, that explicitly gives the behaviour of the nor-gate constructed by wiring the output, `c`, of the or-gate to the input, `c`, of the not-gate.

The only code we have been able to write so far for this uses "reflection", i.e., constructs Aldat statements which Aldat then executes. This is an inelegant facility (and dangerous if it allows mischievous code to be slipped past the static checking mechanisms) and we have not implemented it. I do not show this code here because I am dissatisfied with it. We may need more metadata facilities than we so far have. Or the problem may be a red herring, and better solved with nested relations. It is worth raising here as a motivation to others to go further, and as a simple illustration of what metadata means. Later in the paper I use metadata successfully in a number of different ways.

3. Definitions and Working Example

Examples are best for new ideas unfamiliar to readers, but I will suppose familiarity with classical relations and give a few definitions to highlight some important aspects.

A relation is a subset of the Cartesian product of its domains,
where a *domain* is a set of values. A domain is associated with one or more *attributes*, which are named components of one or more relations. The element of the Cartesian product is called a tuple, and so a relation is a set of tuples. Although Aldat, as a language, has no concept of "tuple", a definition is useful.

A tuple is a mapping from a set of names to corresponding values.
The names are the attributes and the values may change in time, but at any one time, a tuple produces only one value for a given attribute. (This is not to say that such values cannot be structured—tuples, sets, or even relations, for instance.) This definition is significant because it parallels definitions from programming languages and operating systems. The central interest of our work has been integrating programming language and secondary storage using relations, so any parallels among the three we can find are valuable. Here are definitions of scopes (from programming languages) and directories (from operating system management of secondary storage).

A scope is a mapping from a set of names to corresponding values. The names are the language variables.

A directory is a mapping from a set of names to corresponding values. The names are the files contained in the directory. Since an obvious way of representing relations is as files, it makes sense to represent a database as a directory. Thus a database has the same definition as a tuple.

Both of these parallel definitions tell us something which we shall exploit when we come to nested relations. The domain algebra (Section 4) will supply the mechanisms for this.

I will not emphasize formal definitions in this paper, but will use examples. A central example is the following.

PartOf(	*A*	*S*	*Q*)
	`wallplug`	`cover`	1
	`wallplug`	`fixture`	1
	`cover`	`screw`	2
	`cover`	`plate`	1
	`fixture`	`screw`	2
	`fixture`	`plug`	2
	`plug`	`connector`	2
	`plug`	`mould`	1

This relation represents the bill of materials discussed in Section 2 and shown in Figure 2. The attributes are A, S and Q. A stands for *assembly*, S for *subassembly* and Q for *quantity.* A and S could be drawn from the domain of strings, or from some more restricted domain of one-word strings describing parts and assemblies. Note that A and S must be from the same domain. Q could have the domain of positive integers.

Another relation could give the costs, C (domain: reals ≥ 0.01), of buying the raw materials or of putting together a (sub)assembly.

Cost(	*Part*	*C*	)
	`wallplug`	0.04	
	`cover`	0.10	
	`fixture`	0.03	
	`plate`	0.06	
	`screw`	0.05	
	`plug`	0.01	
	`connector`	0.02	
	`mould`	0.08	

Part has the same domain, of course, as A and S, above.

Note a few things about this. The attribute, *Part*, is a *key* of *Cost*, because each value for *Part* uniquely identifies a tuple[2]. But then so is C, by the same reasoning. It is apparent to a database designer that C is only accidentally a key, not intentionally, but this is not easily apparent to any program. The language I discuss has no concept of keys, of the related "functional dependences", or of any other semantic (as opposed to syntactic) distinction. This is for reasons of simplicity, because semantic notions are completely open-ended, so, once started on trying to capture semantics in syntactic notation, it is impossible to stop. We prefer to consider Aldat as an uninterpreted formalism, much as differential equations are not written differently when describing heat conduction or describing a wave function. We leave semantic notions to the database design process (which often can itself be expressed using Aldat).

The second thing to note is that we could specify a manufacturing database consisting of these two relations: *Manufacturing* is the tuple (database) (*PartOf, Cost*). Some theorists have considered it helpful to combine all the relations of any multi-relation database into a single, "universal relation". That cannot be done, without much awkwardness, in the case of *Manufacturing.*

The bill-of-materials example will carry us through Section 6 of the paper. Section 7 will use some new examples, and Section 8 will return to the family tree example of Section 2.

4. Operating on attributes independently of relations: the domain algebra

The "domain algebra" operates on the attributes of relations and satisfies the principle of closure by producing new attributes. (It should perhaps have been called the "attribute algebra", but it wasn't.) It also satisfies the principle of abstraction in that its operations are independent of their context, i.e., of any particular relation. This latter enables the programmer to divide problems

[2]On being told, say, `connector`, *Cost*() can respond only with the single tuple (`connector, 0.02`).

into separate concerns or aspects and solve these more or less independently of each other.

A domain algebra statement behaves like the declaration of a parameterless function. It creates a prescription for future execution. Its result is always a single attribute, which is called a *virtual* attribute because it is not an attribute of any relation. (Normal attributes can be called *actual* attributes. We must wait until Section 5.3 before finding out how virtual attributes can be *actualized* by using the relational algebra.)

The domain algebra consists of either scalar operators or aggregation operators. Domain expressions are open-ended and flexible, but very straightforward, apart from the subtlety that the resulting attributes are virtual. The scalar operators include everything one might expect to be able to do with arithmetic, logic, or string operations. We will even find, when we investigate nested relations (Section 6), that scalar operators also include the relational algebra.

A few examples should convey the idea.

let QQ' **be** $Q * Q'$;

(See the bill of materials in Section 2. Note that $'$ is legal in Aldat identifiers, so QQ' is an identifier.)

let QC **be** $Q * C$;

(See the bill of materials in Section 3. That Q and C are attributes of different relations does not concern the domain algebra.)

let *distance* **be**
sqrt((*x2*−*x1*)**2 + (*y2*−*y1*)**2);

(Built-in functions.)

let *AS* **be** *A* **cat** `" component is "` **cat** *S*;

(Concatenation of strings.)

let *expensive* **be** $C \geq$ `0.05`;

(Creation of boolean.)

let *cheap* **be not** *expensive*;

(Operation on boolean.)

let *costCategory* **be**
if *expensive* **then** `"high"` **else** `"low"`;

(Conditional expression.)

let Q' **be** Q;

(Special case: renaming.)

let *One* **be** 1;

(Special case: constant (same value for all tuples of any relation it is actualized in).)

Aggregation domain operators come in two families, **red**uction and **fun**ctional mapping. These each have variants, **equiv**alence reduction and **par**tial functional mapping, respectively. Where the scalar operations were confined to work within each tuple independently of all other tuples, the aggregation operators combine tuples. Reduction combines all tuples and so permits summing, counting, anding, finding the maximum, etc. (It is stolen from APL.)

let *total* **be red** + **of** C;

(The **red** "metaoperator" applies + to the C value of every tuple.)

let *count* **be red** + **of** 1;
let *maximum* **be red max of** C;
let *allCheap* **be red and of** *cheap*;
let *someCheap* **be red or of** *cheap*;

Aggregations may be combined with scalar operations:

let *average* **be** *total*/*count*;

or, more directly:

let *average* **be**
(**red** + **of** C)/(**red** + **of** 1);

More elaborately:

let *standardDeviation* **be**
sqrt((**red** + **of** C**2)/*count* −
(*average*)**2);

The binary operators, +, **max**, etc., are themselves operands of **red**. Any associative and commutative binary operator may be used by **red**. This includes multiplication (for finding global products or geometric means). It also includes operations of the relational algebra, such as natural join and outer join, as we will see when we look at nested relations. This extension will require us to provide an infix syntax for such operators.

An operator which is non-associative, such as absolute difference, or non-commutative, such as concatenation, will fail to produce a consistent answer in the relational context, in which, because relations are mathematical sets, the order of tuples does not matter. Thus, subtraction, division and other such operations are undefined for reduction.

The important question, "what does reduction do with the result", has already been answered by closure. Since the result is an attribute, it will evidently be a *constant* attribute, with the same value for any tuple in any relation it is actualized in. When writing code in the domain algebra, we do not need to think about this, but eventually the virtual attribute will be actualized in some relation(s), and so it is useful to be able to visualize all those tuples, each with an extra attribute and all having the same value, namely the aggregate. Bear in mind that this is a visualization and not a commitment ever to materialize such redundant values.

Equivalence reduction adds a **by** clause to reduction, and so allows aggregations, such as sums, over groups of tuples. One or more attributes may appear after the **by**, and their values serve to decompose the relation into equivalence classes of tuples, hence the name. One example, from the bill of materials in Sections 2 and 3, suffices to show the extension.

let Q'' **be equiv** + **of** $Q * Q'$ **by** A, S';

This sums, over all pairs of edges with the same first and last vertices but different midpoints, the product of the quantities from the two edges in the pair. (Network theory has variants of this, in which, for instance, + becomes **max** and * is replaced by +.)

Functional mapping permits similar aggregations, but for any binary operator, because it uses the values of one or more attributes to induce an ordering on the tuples. Here is the cumulative value of costs, in ascending order of the costs themselves (bill of materials in Section 3)

let *cumCup* **be fun** + **of** C **order** C;

and here is the accumulation in descending order of C.

let *cumCdown* **be fun** + **of** C **order** $-C$;

Because the semantics is less obvious than the semantics of scalar and reduction operations, I will define it. This definition is best done by algorithm, so we stray ahead of ourselves into actualization for this case.

Here again is the *Cost* relation from Section 3, augmented by values for the two virtual attributes I have just declared. Because they are virtual, and could be actualized for any relation containing the **real** C, these attributes are written outside the parentheses. This reminds us that we are visualizing, not materializing. I have ordered the tuples according to C, to make the process easier to see.

Cost				
(*Part*	C	)	*cumCup*	*cumCdown*
plug	0.01		0.01	0.39
connector	0.02		0.03	0.38
fixture	0.03		0.06	0.36
wallplug	0.04		0.10	0.33
screw	0.05		0.15	0.29
plate	0.06		0.21	0.24
mould	0.08		0.29	0.18
cover	0.10		0.39	0.10

The following algorithm presupposes that the tuples are processed in the order induced by the attribute after **order** (e.g., C or $-C$), that the value in the current tuple of the attribute after **of** (e.g., C) is *value*, that the operator after **fun** is *op*, and that there is an accumulator, *accum*. Finally, the resulting value for each tuple is just the current value of the accumulator.

```
(Initialize) For the first tuple, accum <− value.
(Iterate) For each subsequent tuple,
    accum <− value op accum.
```

We can see that, if *op* is + or **max**, for instance, this will just give the expected cumulative sum or maximum. If *op* is − or / (division), the result will be a cumulative alternating sum or product.

This algorithm does not give the full story. Functional mapping is defined so that, in

let g **be fun** *op* **of** f **order** x;

the functional dependence $x \rightarrow f$ is expected in the data, and the functional dependence $x \rightarrow g$ is generated in the result. (A *functional* in mathematics maps functions to functions.) Thus, if C, above, had any repeated values, the result would also repeat.

Cost

(*Part*	*C*	)	*cumCup*	*cumCdown*
`plug`	`0.01`		`0.01`	`0.33`
`connector`	`0.02`		`0.03`	`0.32`
`fixture`	`0.03`		`0.06`	`0.30`
`wallplug`	`0.04`		`0.10`	`0.27`
`screw`	`0.05`		`0.15`	`0.23`
`plate`	`0.05`		`0.15`	`0.23`
`mould`	`0.08`		`0.23`	`0.18`
`cover`	`0.10`		`0.33`	`0.10`

The language has no provision for reporting if the expected dependence, $x \rightarrow f$, is violated. If x is the same in more than one tuple, the value of f from one of these tuples is chosen, nondeterministically, to calculate g for all of them.

We can use this property to define an idiom which calculates the number of different values of an attribute in a relation. For instance, A in *PartOf* in Section 3 has four different values, which can be counted by

let *diffAs* **be red max of**
  fun + **of** 1 **order** A;

The functional mapping will accumulate 1 only for different values of A, so *accum* will be 1 for `cover`, 2 for `fixture`, 3 for `plug` and 4 for `wallplug`. Then the **red max** will find and return the maximum of these, 4.

Partial functional mapping extends functional mapping to allow grouped accumulations in the same way that equivalence reduction extends reduction. The **fun** keyword becomes **par** and we add a **by** clause, which can name any number of attributes and can be written before or after the **order** clause. If functional mapping can be thought of (when + is used) as a crude form of integration, partial functional mapping is the corresponding partial integration. Mathematicians say that integration is an example of a "functional", a function which maps a function to another function, hence the name of this category of aggregation.

The domain algebra had already been around for a decade when it was first published in 1988 [51]. The **red** and **equiv** reductions contain the aggregations and group-by aggregations of SQL. The most important failure of SQL is not separating such operations on attributes from the operations on relations.

5. A slight elaboration of the relational algebra

5.1. Binary operators

The binary operators of the classical relational algebra are the natural join, the natural composition and division [15,17], as well as the usual set operators of union, intersection, difference, etc. A little insight reveals that these all group into two families which extend them slightly. If we start with binary operations on sets, we have two categories: those that produce sets as a result (union, intersection, etc.) and those that produce booleans (subset, superset, empty intersection, etc.). The first category can be extended to relations in such a way that set intersection extends to the natural join. Then set union gives what later came to be called the outer join, and some new operators appear, based for example on difference and symmetric difference of sets.

The category of boolean-producing set operators can be extended to relations to give natural composition (from not-empty intersection) and division (from the superset test), and some new operators based on subset, empty intersection, etc.

On this basis, I define the μ-join and the σ-join families of relational binary operators. To keep the definitions readable, I suppose that we are joining two binary relations, $R(X, Y)$ and $S(Y, Z)$, with common (join) attribute, Y. These definitions are easily extended to cases where each of X, Y and Z are whole sets of attributes, and to cases where the join attribute does not have the same name in both relations and the correspondence must be put explicitly into the syntax for the join. I define distinct names for each operator, which can be written in infix mode between the two operands, an adjustment of conventional syntax in keeping with the requirement I anticipated in the discussion of the domain algebra (Section 4).

Mu-joins. The μ-joins correspond to the set-valued binary operators on sets.I make three preliminary definitions, of *Gauche, Centre* and *Droit.*

$$G \triangleq \{(x, y, \mathcal{DC}) \mid (x, y) \in R \textbf{ and } (y) \notin [Y] \textbf{ in } S\}$$

$C \triangleq \{(x, y, z) \mid (x, y) \in R$ **and** $(y, z) \in S\}$
$D \triangleq \{(\mathcal{DC}, y, z) \mid (y) \notin [Y]$ **in** R **and** $(y, z) \in S\}$
where
$[Y]$ **in** $S \triangleq$ the usual projection of a single attribute, Y, from the relation S;
and similarly for $[Y]$ **in** R.
Here, $\mathcal{DC}$ is the "don't care" null value. I will not dwell on null values in this paper, but briefly mention that this null value participates in all operations as if it were not there (in particular, it is the left and right identity of any type-preserving binary operator, e.g., $\mathcal{DC} + x = x = x + \mathcal{DC}$).

Now, assuming further only set union, $\cup$, I can define the μ-joins.

Join	*Meaning*	*Definition*
R **ijoin** S R **natjoin** S	natural join, intersection join	C
R **ujoin** S	outer join, union join	$G \cup C \cup D$
R **sjoin** S	symmetric difference join	$G \cup D$
R **djoin** S	difference join	G
R **drjoin** S	right difference join	D
R **ljoin** S	left join	$G \cup C$
R **rjoin** S	right join	$C \cup D$

Of these operators, **ijoin**, **ujoin** and **sjoin** are associative and commutative, and **djoin**, **drjoin** and **ljoin**, **rjoin** redundantly form commutative complements. The natural join is by far the most important in theory and practice, and has two synonymous names.

The special cases of ordinary set-valued set operations arise when X and Z are absent (or, equivalently, contain only $\mathcal{DC}$ values), so that both R and S are sets of Y values (*unary* relations). In this special case, the **ljoin** and **rjoin** become superfluous, returning merely the left operand or the right operand, respectively.

The reader familiar with the classical natural join should have no difficulty grasping its extended family of μ-joins without need for examples.

Sigma joins. Table 5.1a defines the σ-joins, where
$R[x] \triangleq$ select $X = x$ in R and then project the result on the attribute(s) other than X; and similarly for $S[z]$.

(Aldat also allows synonyms such as **div** (division) for **sup** (superset), and a single **!** prefixed to any operator name gives the complementary operator. **Sep** (separate), and the set comparison symbol, @, mean empty intersection. The complement, natural composition, has the synonym **icomp**, bacause of its connection with natural join, **ijoin**.)

The set relationships can be reduced to fundamentals using another three auxiliary definitions, of *suBset, sEparate* and *suPerset*,
$B \triangleq \exists y(y \in R[x]$ **and** $y \notin S[z])$
$E \triangleq \exists y(y \in R[x]$ **and** $y \in S[z])$
$P \triangleq \exists y(y \notin R[x]$ **and** $y \in S[z])$
Here, I am comparing two sets on Y, $R[x]$ and $S[z]$. B is true if $R[x]$ is not a subset of $S[z]$, P if $R[x]$ is not a superset of $S[z]$, and E if $R[x]$ and $S[z]$ do not have an empty intersection.

The twelve set comparisons reduce to combinations of these three (Table 5.1b).

The σ-joins are a longer stretch away from classical relational algebra than are the μ-joins, so I provide some examples. The first is classic division, using only the assembly and subassembly columns of *PartOf* from Section 3, and the relation *CovFix*, shown. *Find subassemblies,* S*, which are used by both* `cover` *and* `fixture`
Suba <− *PartOfAS* **sup** *CovFix*;

PartOfAS(	*A*	*S*	)
	`plug`	`connector`	
	`wallplug`	`cover`	
	`wallplug`	`fixture`	
	`plug`	`mould`	
	`cover`	`plate`	
	`fixture`	`plug`	
	`cover`	`screw`	
	`fixture`	`screw`	

CovFix(	*A*	)
	`cover`	
	`fixture`	

Suba(	*S*	)
	`screw`	

Join	*Definition*	*Join*	*Definition*
R **propsub** S	$R[x] \subset S[z]$	R **!propsub** S	$R[x] \not\subset S[z]$
R **sub** S	$R[x] \subseteq S[z]$	R **!sub** S	$R[x] \not\subseteq S[z]$
$R = S$	$R[x] = S[z]$	R **!=** S	$R[x] \neq S[z]$
R **sup** S	$R[x] \supseteq S[z]$	R **!sup** S	$R[x] \not\supseteq S[z]$
R **propsup** S	$R[x] \supset S[z]$	R **!propsup** S	$R[x] \not\supset S[z]$
R **sep** S	$R[x] \Cap S[z]$	R **natcomp** S	$R[x] \not\Cap S[z]$

Table 5.1a: defining the σ-joins.

Set Comparison	*Definition*	*Set Comparison*	*Definition*
$R[x] \subset S[z]$	$\neg B \wedge P$	$R[x] \not\subset S[z]$	$B \vee \neg P$
$R[x] \subseteq S[z]$	$\neg B$	$R[x] \not\subseteq S[z]$	B
$R[x] = S[z]$	$\neg B \wedge \neg P$	$R[x] \neq S[z]$	$B \vee P$
$R[x] \supseteq S[z]$	$\neg P$	$R[x] \not\supseteq S[z]$	P
$R[x] \supset S[z]$	$B \wedge \neg P$	$R[x] \not\supset S[z]$	$\neg B \vee P$
$R[x] \Cap S[z]$	$\neg E$	$R[x] \not\Cap S[z]$	E

Table 5.1b: implementing the σ-joins.

Two more subtle queries find
a) *assemblies except those containing* `plate`*s or* `screw`*s*,
NoPorS <− *PartOfAS* **sep** *PS*;
and, b) *assemblies except those containing* `plate`*s and* `screw`*s*,
NoPandS <− *PartOfAS* **!sup** *PS*;

PartOfAS(	*A*	*S*	)
	`cover`	`plate`	
	`cover`	`screw`	
	`fixture`	`plug`	
	`fixture`	`screw`	
	`plug`	`connector`	
	`plug`	`mould`	
	`wallplug`	`cover`	
	`wallplug`	`fixture`	

PS(	*S*	)
	`plate`	
	`screw`	

NoPorS(	*A*	)
	`plug`	
	`wallplug`	

NoPandS(	*A*	)
	`fixture`	
	`plug`	
	`wallplug`	

(*PartOfAS* is the same relation as in the previous example, but the tuples have been regrouped to make it easier to see how the joins result.)

The final example is natural composition, using the relations *PartOf* and *Cost* from Section 3. *Find assemblies and their associated costs.*
AC <− *PartOf* **icomp** *Cost*;

AC(	*A*	*C*	)
	`wallplug`	`0.10`	
	`wallplug`	`0.03`	
	`cover`	`0.06`	
	`cover`	`0.05`	
	`fixture`	`0.01`	
	`fixture`	`0.05`	
	`plug`	`0.02`	
	`plug`	`0.08`	

Note first that natural composition and **sep**, its complement, are the only σ-joins in which one need not worry about additional attributes. In the other ten σ-joins, we may need to project out extra attributes which could interfere in the grouping of the values, possibly causing some results to be left out.

Note second that the above example for natural composition captures the costs for every subassembly of the assemblies shown in the result (including the 5 cents for `screws` both in `cover` and in `fixture`). If, however, the costs were the same for two subassemblies of one assembly (e.g., `plates` cost 5 cents), then one of the above tuples would disappear (e.g., the 6-cent tuple for `cover`). This shows both the kinship between natural composition and natural join and an example of when it would be better to use natural join to keep the subassembly in the result so as not to lose tuples.

In these examples, the second operand has no non-join attribute. σ-joins are not limited to this special case. An example finds all pairs, *Part* and *Part′*, from relations *PC*(*Part, Colour*) and a renamed copy *PC′*(*Part′*, *Colour*), such that *Part* comes in at least all the colours that *Part′* does:

$PP' <- PC$ **sup** PC';

PC(	*Part*	*Colour*)
	`plate`	`white`
	`plate`	`brown`
	`plate`	`brass`
	`mould`	`white`
	`mould`	`brown`

PP′(	*Part*	*Part′*)
	`plate`	`plate`
	`plate`	`mould`
	`mould`	`mould`

In an extreme special case, both operands have no non-join attributes. Since the σ-joins all eliminate the join attribute from the result, this case results in a relation with no attributes at all, a *nullary* relation. Such a result always comes from comparing two simple sets, and the result must be boolean. We can convince ourselves that a nullary relation is always a boolean by the following argument. Imagine that a nullary relation has tuples, somehow. Since there is no attribute and hence no attribute values to distinguish tuples from each other, we cannot count these tuples: they are either there or they are not. So nullary relations have two possible values, which we can take to be the two possible boolean values. These can be arranged so that the nullary-relation value is `true` if the set comparison being made is true, and `false` otherwise.

σ-joins may be implemented using natural join to determine set intersections, and reduction or equivalence reduction of the domain algebra to count whether or not this intersection encompasses the whole set. Thus, σ-joins are not independent of the rest of Aldat. I leave it as an exercise to show this equivalence.

Before we leave binary operators, we must discuss joins on non-common attributes. A notation is needed to say which attribute is joined to which, and, since the domain algebra demands it, this notation must be infixed. Thus we return to Codd's original join notation which lists the join attributes explicitly, if they are not the common attribute. To join $R(W, X)$ with $S(Y, Z)$ on X and Y, we write $R[X$: <join> :$Y]S$. If there are multiple join attributes, $U[C, D, E$: <join> :$F, G, H]V$ will, for instance, join relations U and V, pairing attributes C and F, D and G, and E and H. (If the join is on common multiple attributes, no special notation is needed.) Some obvious rules apply, for instance that the paired attributes must be compatible ("union-compatible") and that no join must result in attributes of the same name coming ambiguously from both operands.

5.2. Unary operators

In Aldat, the classical operations of projection and selection are fused into a single "T-selector" syntax. This is then generalized to include quantifiers, giving "QT-selectors". "T" in "T-selector" stands for "tuple", and the selection condition in a T-selector must be such that each tuple can, independently of any others, produce a true or false value for the condition, which determines whether or not that tuple appears in the result. QT-selectors, like the aggregation operators of the domain algebra and like the σ-joins, operate on sets of tuples to produce their answers.

Related to T-selectors is **grep**, an operator which uses regular expressions to find matches to patterns in any attribute of its argument, and uses metadata to say which attribute this was and its type, as well as to give the position and the value of the match.

The third category of unary operators we discuss is the *editors*, an open collection of interactive end-user interfaces allowing a relation to be interpreted in specialized ways.

T-selectors The T-selector that finds assemblies that use screws, and the numbers of screws used, in *PartOf* (Section 3) is

$[A, Q]$ **where** $S=$`"screw"` **in** *PartOf*

The special case, projection, to give *PartOfAS* used to illustrate σ-joins in Section 5.1, is

PartOfAS $<-$ $[A, S]$ **in** *PartOf*;

(Note that I have sometimes written whole statements, terminated by a semicolon, with an assignment, $<-$, from the relational expression. Sometimes I have just written the relational expression.) The other special case of T-selectors is selection (without subsequent projection): the projection list, [<attribute list>], is omitted before the **where**, and the result is on all attributes of the operand. [3]

If the projection list is present but empty, [ ] **where** .., or [ ] **in** .., the result is a nullary relation, i.e., a boolean, and the syntax is read "something where .." or "something in ..": it is true if (the selection on) the relational expression ("..") is not empty and false if it is empty.

T-selectors require a single pass of the operand, with sorting to detect duplicates. Many applications need a special case which can be run in sublinear time. We permit a special syntax for T-selectors in which the selection is a conjunction of clauses of the form <attribute>=<constant>, and the projection yields all the other attributes, not mentioned in the selection condition. Thus we write $R[u, v, w, , ,]$ to mean

$[X, Y, Z]$ **where**
  $U = u$ **and** $V = v$ **and** $W = w$ **in** R

for the relation $R(U, V, W, X, Y, Z)$. Note the position dependence, indicated explicitly by the commas: $R[, , , x, y, z]$ means

$[U, V, W]$ **where**
  $X = x$ **and** $y = y$ **and** $Z = z$ **in** R

(Trailing commas may be omitted: $R[u, v, w]$ is the same as $R[u, v, w, , ,]$.)

This shorthand can be taken to be syntactic sugar, since it is defined by the special case of the general T-selector. But it is sugar which deserves its own implementation, using sublinear techniques such as hashing. (This does not identify query optimization with syntactic sugar, or rule out independent optimizations in the implementation, unseen by the programmer.)

QT-selectors, , on the other hand, push the general T-selector to an extreme, given the limits of an implementation costing at most a sort followed by one pass of the operand. QT-selectors count numbers of different values of attributes within groups defined by other attributes, and evaluate predicates based on these counts: these are *quantifier predicates* and they can be used to define quantifiers that go well beyond the classical for-all and for-some.

Since the capabilities of QT-selectors will be new to most readers, I start with examples before giving a definition. Here is the QT-selector

[3]Although the paper will not discuss concurrent programming in Aldat, I can mention the synchronization primitive. This is based on Linda [8], which is tuple-oriented. The **where** in the T-selector is replaced by **when** and the semantics of the result is identical unless the original T-selector would have returned an empty relation. In that case, the **when** causes the T-selector to block until some external change to the operand renders the potential result non-empty. Unlike Linda's, this construct is deterministic: all relevant tuples are returned. (Aldat implements nondeterminism independently with the unary relational operator, **pick**; this returns a singleton relation consisting of one of the tuples nondeterministically chosen from its operand.) To demonstrate the completeness of this construct, in the context of Aldat, I use it to implement a semaphore.

```
relation SEMAPHORE(Sem_name, Sem_count);
comp I(sema) is { SEMAPHORE <+ sema; };
comp P(sema) is
{ update SEMAPHORE change Sem_count <-
   Sem_count − 1 using ([Sem_name]
      when Sem_count > 0 in
   (SEMAPHORE ijoin [Sem_name] in sema));
};
comp V(sema) is
{ update SEMAPHORE change Sem_count <-
  Sem_count + 1 using ([Sem_name] in sema);
};
```

We require only that individual Aldat statements be atomic in their effects. We have used an incremental assignment, *SEMAPHORE* $<+$ *sema*; instead of the longer equivalent, *SEMAPHORE* $<-$ *SEMAPHORE* **ujoin** *sema*; **Update**s will be discussed in Section 5.5 and **comp**utations (procedures) in Section 7.

answering the query *Find subassemblies used in exactly two assemblies.*

[*S*] **quant** (#=2)*A* **in** *PartOf*

The *quantifier symbol*, #, counts the number of different values of *A*, the attribute following the parentheses containing the quantifier predicate #=2. Clearly, this formulation permits an indefinite number of possible quantifier predicates—any predicate we like involving the number #: 1<# **and** #<4 finds 2 or 3 different assemblies (as does #=2 **or** #=3); #**mod** 2=0 finds even numbers of different assemblies, and so on. (The answer to all these queries is `screw`.)

Let's *Find subassemblies used in exactly two assemblies in quantities of less than 3.*

[*S*] **quant** (#=2)*A* **where** *Q* <3 **in** *PartOf*

QT-selectors, like T-selectors, are evaluated from right to left. In the above example, the classical selection, *Q* <3, is done first, followed by the count, #, of the number of different values of *A* and the evaluation of the quantifier (#=2)*A*, followed finally by the projection on *S*.

Here is the algorithm that defines QT-selectors [81,88].

1. (Sort) Group the tuples of the relation on attribute within attribute .. within attribute, working through the attributes from right to left as they appear in the QT-selector and projection list.

2. (One pass) Process the sorted tuples, maintaining counters for each quantified attribute: working the quantified attributes from right to left in the QT-selector, count the number of different values of the attribute for which the predicate to the right is true, and when the attribute to its left changes value, evaluate the predicate and reset the counter. The rightmost predicate, corresponding to no quantified attribute, is the **where** selection condition or, if no **where**, then **true**. The values of the attributes in the projection list give the result whenever the leftmost predicate is **true**.

The right-to-left evaluation in this algorithm tells us how to interpret more advanced QT-selectors, with quantifiers on more than one attribute. For an interesting result, I extend the *Part-Colour* example (Section 5.1) with an attribute Material, *Mat*, so that the brown and white plates come either in enamelled metal or plastic, the brass plate is metal only, and the moulds are plastic only (or else they would have unfortunate electrical properties). Then

[*Part*] **quant** (#=2)*Col*, (#=2)*Mat* **in** *PCM*

has the answer `plate`. Here is the processing laid out, after sorting so as to group *PCM* on *Mat* within *Col* within *Part*.

PCM

(*Part*	*Col*	*Mat*	)	#*Mat*	#*Col*
`mould`	`brown`	`plastic`		1	
`mould`	`white`	`plastic`		1	0
`plate`	`brass`	`metal`		1	
`plate`	`brown`	`metal`			
`plate`	`brown`	`plastic`		2	
`plate`	`white`	`metal`			
`plate`	`white`	`plastic`		2	2

Although this can be executed in one pass, I will describe it in two. First, the number of different materials is counted within each *Part-Col* group. That number is assigned to # and the *Mat* quantifier predicate, #=2, evaluated: it is true only for (`plate`, `brown`) and (`plate`, `white`). Next, the number of different colours is counted within each *Part* group, but the contribution to the count is zero for any false value of the *Mat* quantifier predicate. The *Col* quantifier predicate, #=2 (again, coincidentally), and is true only for `plate`. Thus, `plate` is the answer.

If we reverse the order of the quantifiers, we get a different answer: no *Part* satisfies the query. This is not necessarily wrong. Reversed quantifiers classically give different answers: $\forall x \exists y (y > x)$ is not the same as $\exists y \forall x (y > x)$. We work through the evaluation to confirm.

[*Part*] **quant** (#=2)*Mat*, (#=2)*Col* **in** *PCM*

PCM

(*Part*	*Mat*	*Col*	)	*#Col*	*#Mat*
`mould`	`plastic`	`brown`			
`mould`	`plastic`	`white`		2	1
`plate`	`metal`	`brass`			
`plate`	`metal`	`brown`			
`plate`	`metal`	`white`		3	
`plate`	`plastic`	`brown`			
`plate`	`plastic`	`white`		2	1

This time, we evaluate the number of colours first, then, for the counts that equal 2, we evaluate the number of different materials. That quantifier predicate is not satisfied anywhere, so the answer is empty.

This difference is bad news for natural-language querying, without feedback from the query system. A native English speaker has difficulty distinguishing *Find parts that come in two different colours and are made from two different materials* from *Find parts that are made from two different materials and come in two different colours.* It is extreme to expect a computer program to make the distinction.

For simple queries, the natural language maps almost directly into the QT-selector. But we see that QT-selectors rapidly become too sophisticated for straightforward interpretation.

It is possible to write an implementation of all QT-selectors, except those with a predicate #=0, using T-selectors and equivalence reduction from the domain algebra. An Aldat implementation may not execute this in one pass, but it will not need to do more than one sort. This is an exercise for the reader.

There is a second quantifier symbol, •, which is read "proportion of": $\bullet \triangleq$ #/<some count>. The count on the denominator of this definition could be made in several ways, but Aldat counts the number of different values of the attribute being quantified in whatever relational expression follows the **in** of the QT-selector. In this way, we can express "for all" and "most", among other quantifiers.

Expressions of the domain algebra may appear in a quantifier predicate, increasing flexibility even more.

QT-selectors appeared in 1978 [48].

Grep. Queries should not always oblige the programmer to specify a particular attribute in advance. Aldat provides a pattern-matching facility which can apply a regular expression to the entire relation that is its operand and return the position and value of the match, and the attribute and its type that contains the match. This is called **grep**, after the Unix command, which here stands for "get regular expression pattern" (although this is only one of the possible historical translations [29]).

The basic **grep** can be illustrated by finding the substring "`plug`" in the *PartOf* example from Section 3.

Plugs <− **grep**(*attr, type, pos, value*) `"plug"` **in** *PartOf*;

The result is shown in Table 5.2a.

We see that **grep** returns the whole tuple if a match is found anywhere in that tuple. **Grep** generates the additional attributes named in its parameter list. There may be up to four of these (if none, the whole parameter list may be omitted) and **grep** recognizes them by *type*, not by name or position in the list. The above example expects the prior declarations

domain *attr* **attrib**;
domain *type* **type**;
domain *pos* **intg**;
domain *value* **strg**;

and the type of each attribute, whether the ordinary types, **intg** or **strg**, or the metadata types, **attrib** or **type**, tells **grep** how to use it.

In the example, the **type** attribute, *type*, is unrevealing, but in general **grep** will recognize patterns in types of data other than strings, such as integers, reals or even booleans, and so a **type** attribute is useful. Similarly, the **strg** attribute, *value*, is uninformative, because the example uses only substring match. When a regular expression is used, an attribute such as *value* tells us what (sub)string satisfied it. In the example, the position, *pos*, is counted from 0 at the first byte of the attribute.

If a pattern is matched more than once in a tuple of the operand, extra tuples may be generated in the result, one for each match. Depending

Plugs(*A*	*S*	*Q*	*attr*	*type*	*pos*	*value*)
`wallplug`	`cover`	`1`	`A`	`strg`	`4`	`plug`
`wallplug`	`fixture`	`1`	`A`	`strg`	`4`	`plug`
`fixture`	`plug`	`2`	`S`	`strg`	`0`	`plug`
`plug`	`connector`	`2`	`A`	`strg`	`0`	`plug`
`plug`	`mould`	`1`	`A`	`strg`	`0`	`plug`

Table 5.2a: basic **grep**.

on what attributes are specified in the parameter list, some of these tuples may be duplicates in the result, and so will not appear.

A refinement of **grep** adds a second list of **strg** parameters, separated from the above list by a semicolon. The names in this list must also appear in the pattern to be matched, and are treated as wildcards with the same role as `".*"` in the pattern. They are used to return parts of the values that matched the pattern.

MorePlugs <− **grep**(*attr, pos; x, y*)
`"\xplug\y"` **in** *PartOf*;

The result is shown in Table 5.2b. These wildcards are always strings. Here, y is the nullstring in every tuple because nothing follows "..`plug`" in the matching attribute. Similarly for x in three tuples. Note that *pos* is `0` in every case, because the pattern now is `"\xplug\y"` and this starts at the beginning and ends at the end of every match.

Editors. In addition to T-selectors, QT-selectors and **grep**, Aldat can support an open-ended collection of relational "editors". These are unary operators with two faces. One faces the end-user, and provides a language of interactive commands, such as changing, adding or deleting a tuple, moving to the next tuple, and saving the result. The other faces the programmer ("programmer-user"), and looks just like an algorithmic unary operator, such as a T-selector; except, instead of an algorithm, the end-user does the work.

A simple example is an editor based on any well-established text editor, such as `vi`. The programmer-user would have expression syntax such as

[A] **vedit** R

which would open the data in R to be processed by the end-user and then incorporate the result into a containing expression and eventually a statement, e.g.,

T <−[A, C] **in** S **ijoin** [A] **vedit** R;

The end-user sees none of this, but is presented with an ASCII file containing relation R (say with one line per tuple) sorted on attribute A and with the usual editing commands for `vi`. When the manual work by the end-user is finished and `:wq` entered (the `vi` write-and-quit command), the result is saved, not to R but to the intermediate value of the expression, and control is returned to the statement in the programmer-user's program. This is an example of a general editor, which can accommodate any relation.

We have implemented a general relational editor, with a tuple-oriented interactive language such as the above. We have also implemented specialized editors which interpret their relation operands as, for example, a fact-base for logic programming [30] (the interactive language is Prolog commands), a map representation for a geographic information system [11] (the editing language is based on commercial G.I.S. interactive interfaces), and so on. The only limit is the imagination.

Here, I discuss a two-dimensional *display* editor, which is a first step towards a general graphics interface, in which any relation has a graphical aspect and any graphical construct has a relational aspect. The key to such a construction is a vocabulary which assigns, to the attributes of the relation, graphical meanings, such as Cartesian abscissa or ordinate, polar radius or angle, linestyle, or colours for point, line or fill. This keeps the relations and their graphical interpretations independent of each other: the relations are ordinary relations, subject to all the operations we have been discussing, and we need not specialize any relation for graphical display. The

MorePlugs(*A*	*S*	*Q*	*attr*	*pos*	*x*	*y*)
wallplug	cover	1	A	0	wall	
wallplug	fixture	1	A	0	wall	
fixture	plug	2	S	0		
plug	connector	2	A	0		
plug	mould	1	A	0		

Table 5.2b: advanced **grep**.

vocabulary can be contained in a two-attribute relation. Table 5.2c gives an example, which tells the display to treat any attribute named *x1, x2* or *x3* as the abscissa of a Cartesian coordinate system, *y1, y2* or *y3* as the ordinate, and so on.

The display syntax that uses this on an ordinary relation, *Show*, is

display2D (*Vocab*) *Show*

This syntax, just like the syntax for a T-selector or a QT-selector or **grep**, gives a relational expression, which can be joined or selected or grepped or assigned to a result in a statement. The value of this relational expression is, so far, simply the value of *Show*.

For example, *Show*(*x1, y1*) might be the relation, and the vocabulary in *Vocab* would cause **display2D** to draw a scatterplot of black points. If *x1, y1* are numeric, they give the coordinates. If *x1, y1* are not numeric, the locations are given by cardinals in ascending order of the attribute values. Table 5.2c also shows, conceptually, a numeric example. Zhu's thesis [91] gives fuller treatment.

Since **display2D** is an editor, the end-user may make changes, using an interactive language supplied. These changes become reflected in the value of the **display2D** expression, so that it will no longer have the value of *Show*. Note that *Show* itself does not change as a result of the updates. Updates will not be allowed, however, which attempt to add attributes to the **display2D** expression. Thus, changing the colour of an edge from the default may require a **linecolour**-linked attribute to be added, and this will fail. Attempting to change the colour of one of the edges of a triangle will fail, unless the triangle was represented in the first place as a polygon. The end-user must communicate with the programmer so

Vocab

(*attr*	*role*	)	comment
x1	cart1		Cartesian abscissa
x2	cart1		
x3	cart1		
y1	cart2		Cartesian ordinate
y2	cart2		
y3	cart2		
rad	polar1		polar radius
ang	polar2		polar angle
a	cart1show		show value of a at position given by a
b	cart2show		
sq	sequence		sequence (for polylines)
ls	line_style		e.g., 0=solid, 1=dashed
lt	line_thick		in units of 1/80 inch
lc	line_colour		e.g., 0=black
tc	text_colour		
fc	fill_colour		
fp	fill_pattern		
:			

Show(*x1*	*y1*)
5	3
3	5
7	7

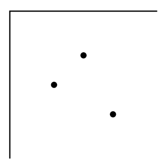

Table 5.2c: Vocabulary for **display2D** and **display2D** (*Vocab*) *Show*

the latter provides a relation which will accommodate all the intended changes. (A fully polymorphic treatment of relations would overcome these limitations, but we are not there yet.)

Display2D achieves still more flexibility by supporting nested relations as its operand. I must omit this discussion.

5.3. Virtual attributes and the relational algebra

I have so far discussed the relational algebra in terms of actual attributes. The attributes that appear in any operator so far are attributes of the relations that also appear. What about virtual attributes?

This is where domain algebra and relational algebra meet. A virtual attribute may appear anywhere in the above operations that an actual attribute does, and that is exactly how virtual attributes get actualized and subsequently appear in some relation(s). The most likely way to actualize an attribute is in a projection list. For example, given the join of *PartOf* with *Cost* (Section 3), here is the actualization of *QC*, defined in Section 4 to be the product of *Q* and *C*.

PartCost <− [*A, S, QC*] **in**
  (*PartOf* [*S*: **ijoin** :*Part*] *Cost*);

Recall that the domain algebra defining *QC* was

let *QC* **be** *Q* * *C*;

By convention, we may write a virtual attribute outside the parentheses containing the actual attributes of a relevant relation, and its possible values below it for each tuple. Table 5.3 shows the natural join of *PartOf* and *Cost* as specified above, with *QC* alongside. (Attributes *S* and *Part* are made aliases by the join.) (I have also shown a second virtual attribute, defined

let *tot* **be red** + **of** *QC*;

which I shall come to shortly.)

The projection that actualizes *QC* gives

PartCost

(*A*	*S*	*QC*)
`wallplug`	`cover`	`0.10`
`wallplug`	`fixture`	`0.03`
`cover`	`screw`	`0.10`
`cover`	`plate`	`0.06`
`fixture`	`screw`	`0.10`
`fixture`	`plug`	`0.02`
`plug`	`connector`	`0.04`
`plug`	`mould`	`0.08`

Now let's look at *tot*. The way it appears as a virtual attribute of *PartOf* [*S*: **ijoin** :*Part*] *Cost* seems to be redundant and wasteful. But it is not, because *tot* is a virtual attribute at that point, and no values are stored anywhere. This is just a way of *thinking* about, or visualizing the virtual attribute. It is essential to think of it as having a value for each tuple of any relation we may want to associate it with, because, by the principle of closure, it is an attribute, and attributes *have* values for each tuple. Just because *tot* is the result of a reduction and so has only one value does not change this.

The smart programmer will actualize *tot* in a non-redundant way.

TotalQC <− [*tot*] **in**
  (*PartOf*[*S*: **ijoin** :*Part*] *Cost*);

This gives a relation which is *singleton* (only one tuple) and *unary* (only one attribute).

TotalQC(*tot*)
`0.53`

TotalQC almost looks like a scalar, the number `0.53`, but it is not. It is a relation. However, we already found out that some scalars, namely booleans, are relations—nullary relations. So there is some commerce between relations and scalars, and we might as well make this complete so that the result of a reduction such as the one that made *tot* can be a scalar, too.

A simple insight permits this without new syntax. I created *TotalQC* as a projection on the attribute *tot*, which was defined as a reduction. The result of any reduction is a constant attribute—it has the same value for all tuples—and so judicious projection will give a singleton, unary relation, like *TotalQC*. What makes *TotalQC* a relation is that *tot* has a name, and this

PartOf [*S*: **ijoin** :*Part*] *Cost*

(*A*	*S*,*Part*	*Q*	*C*)	*QC*	*tot*
wallplug	cover	1	0.10	0.10	0.53
wallplug	fixture	1	0.03	0.03	0.53
cover	screw	2	0.05	0.10	0.53
cover	plate	1	0.06	0.06	0.53
fixture	screw	2	0.05	0.10	0.53
fixture	plug	2	0.01	0.02	0.53
plug	connector	2	0.02	0.04	0.53
plug	mould	1	0.08	0.08	0.53

Table 5.3: join with virtual attribute.

name is the name of the attribute of *TotalQC*. If the attribute of *TotalQC* had no name, *TotalQC* could not very well be a relation. So the secret is to make the attribute *anonymous*. Together with having the result unary and singleton, anonymity can be taken to lead to a scalar. Here is the code (no new syntax).

QCtot <−
[**red** + **of** *QC*] **in** (*PartOf* [*S*: **ijoin** :*Part*] *Cost*);

This defines a scalar real (since *QC* is real and hence so is **red** + **of** *QC*), *QCtot*, with value 0.53.

This operation is called "level-raising", because it can be thought of as bringing an attribute up out of a relation and giving it an independent life of its own.

Because of level-raising, scalars must be treated alongside relations in a database. We will see, when we come to nested relations, that the opposite direction is also taken: relations must be treated alongside scalars in any tuple. Our identical definitions of database and tuple in Section 3 begin to bear fruit.

5.4. Views and recursion

While statements of the domain algebra are merely definitions, like specifying a parameterless function, all the statements we have seen so far in the relational algebra produce data when executed. These have been assignment statements, using <−, and we have seen one incremental assignment, using <+.

The analogue, in the relational algebra, of specifying a parameterless function is the *view*. We need syntax to replace the assignment operator, <−, if we want to define views, and we use the keyword **is**. Anywhere we have so far used <− we can now alternatively use **is**. The effect will be to produce no data but to hold off evaluation until some subsequent assignment, using the view defined by the **is**, is executed.

This is all straightforward and useful in the conventional way. It becomes really interesting if the view is *recursive*. An example is the classical definition of ancestor in terms of parent. In the relational algebra, given a relation *parent*(*sr*, *jr*), we can define *ancestor* as a recursive view in a way which closely parallels

An *ancestor* **is** a *parent* or
the *parent* of an *ancestor*:
ancestor **is** *parent* **ujoin**
(*parent*[*jr*: **icomp** :*sr*] *ancestor*);

Such a recursive view was used in the bill-of-materials processing shown in Section 2. The view I gave is

Explo **is** $[A, S, Q]$ **in** $[A, S, Q''']$ **in**
(*PartOf* $[A, S$ **ujoin** $A, S']$ $[A, S', Q'']$ **in**
(*Explo* $[S$ **natjoin** $A']$ $[A', S', Q']$ **in**
PartOf));

This parallels the *ancestor* view but uses four projections to actualize domain algebra. Some of this actualization is renaming, and the most obvious difference between this view and the *ancestor* view is that it uses a natural join instead of a natural composition. This is because the projection implicit in natural composition, which gets rid of the join attribute, may in this case cause us to lose tuples which we will need in doing the sum. (I gave an example in Section 5.1 of this risk of using natural composition.) Some of the renaming is needed because the natural join does not do this projection and would produce a result with different attributes having the same name, an unacceptable ambiguity. All the single-primed attributes are renamings, as we saw in Section 2.

The calculation which sums products of quantities, so that all paths of two edges connecting the same endpoints contribute to the total quantity, is that of matrix multiplication.

let Q'' **be equiv** + **of** $Q * Q'$ **by** A, S';
The benefit of the independence of the domain algebra from the relational algebra is that we need here think only of two-edge paths. The relational recursion will automatically consider this resulting path to be a single edge in the next step.

One further consideration must be made, which is to bring together the quantities from paths of different lengths.

let Q''' **be** $Q + Q''$;
This is actualized after taking the union.

Finally a subtlety.

let Q **be** Q''';
This renaming seems to be recursion in the domain algebra, since Q''' is defined in terms of Q. It is not. Such recursion would be illegal (and Aldat will warn the programmer about the cyclic definition) because the actualizer cannot generally know whether the original Q or the newly-defined virtual Q is intended. However, the above definition of *Explo* is unambiguous, because the new Q is projected from a relation which does not contain Q, so the Q projected must be the virtual one. The purpose, of course, is to rename Q''' so that *Explo* is defined on the correct attributes for the next recursive step.

I must say a word about the implementation of recursive views in the relational algebra. The view is replaced by a loop containing the expression after the **is** keyword. The target relation is initialized to empty, and the loop continues until there is no more change in the target relation. This can give a highly inefficient implementation of the transitive closure for *ancestor*, for instance, but it is the most flexible interpretation of the recursive view in general. Significantly, this ultra-naive implementation of recursion can be used by a sophisticated programmer to compute, say, transitive closures with maximum efficiency [13,14]. (It is also possible for a programmer to define recursions that do not stop. Aldat is not intended to be an over-protective language.)

Finally, I can remark on a possible fusion of domain algebra and relational algebra syntax. The **is** keyword defines a parameterless function on relations in just the same way that **let**...**be** defines a parameterless function on attributes. In Section 6 I am going to dissolve the distinction between relations and attributes. So it makes sense to fuse the two syntactic notations and replace, say, **let**...**be** by **is** everywhere. I will, however, not do that in this paper, because it still seems useful, and reduces confusion, to be able to spot domain algebra through its distinctive syntax.

5.5. Updates

Expressions of the relational and domain algebras are purely declarative (sometimes called "functional"), in the technical sense of programming language theory, that results depend only on the explicitly-given inputs and there are no side effects. Assignments and definitions of views and virtual attributes introduce state, i.e., side effects. Inevitably, in a language intended to process large amounts of data, we must have a more localized way of effecting changes, or else be faced with having to copy whole relations. Thus, as in any database language, we need updates.

Even though we introduce an update mechanism to make local changes, we must avoid the trap of thinking in terms of tuples. Aldat is intended to abstract away from tuples. For this reason all our operators so far abstract over looping. So updates will work with sub-relations, but will not force the programmer to think about tuples.

The central way to achieve this is to use relations to control update. An alternative is to use predicates, but, as we have seen, QT-selectors and joins support a tremendous variety of predicates, so this alternative is largely subsumed under control by relations. For this reason, we consider updates in the context of the relational algebra.

As an example, lets consider various ways to change the cost of `plate` from 6 cents to 5 cents in *Cost*(*Part, C*) (Sections 3, 4 and 5.1). We start with the full syntax for **update**..**change**, using the relation

```
NewCost(Part   C′)
        plate  5
```

update *Cost* **change** $C <\!\!- C'$ **using**
ijoin on *NewCost*;

This computes the natural join of *Cost* and *NewCost*, flagging the tuples of *Cost* that participate in the join, and then replacing the value of C by the values of C' in those tuples.

Joins other than natural join may be used, but they might not make sense here. For instance, every tuple of *Cost* participates in the outer join with *NewCost*, so **using ujoin** in the above statement would flag every tuple for changing. Fortunately, the value of C' is $\mathcal{DC}$ for all of these tuples except those whose *Part* is `plate`, and this null value is supposed to have no effect. So the assignment, $C <\!\!- C'$, is designed not to change C should C' be the $\mathcal{DC}$ null. Thus **using ujoin** gives the same result here as **using ijoin**. I leave the reader to come up with examples in which **ujoin** and **sjoin** produce results which are different from **ijoin**, and each other, but useful. [Hint: a frequent update operation is "upsert": update an item if the item is in the natural join of *Cost* and *NewCost* else insert the item from NewCost. Ed.]

Since **ijoin** is the join most likely to be used, it is the default.

update *Cost* **change** $C <\!\!- C'$
using *NewCost*;

is identical to the explicit **ijoin** update above.

The statement between **change** and **using** may be replaced by any number of statements, separated by semicolons and enclosed in curly brackets. Thus any number of attributes may be changed in the flagged tuples. Note that the right-hand sides of these assignments may be any domain algebra expression. This is the first time we have seen the assignment operator, $<-$, in connection with the domain algebra.

There are many other ways to write the update intended by the above within the syntax given. These all involve writing different relational expressions instead of *NewCost*: projections, joins, QT-selectors, editors, etc. Since the update in the example is so simple, we could also use truncated syntax with the predicate expressed in the domain algebra instead of the relational algebra.

update *Cost* **change** $C <\!\!-$
if *Part* = `plate` **then** `5` **else** C;

This flags and updates every tuple of *Cost*, but in all the non-`plate` tuples C is replaced by itself so no change is apparent. Clearly it is likely more efficient to use the **ijoin**.

Accompanying **change** are **add** and **delete** syntax. These just give effects which we can already achieve with assignment and the relational algebra.

update *Cost* **add** *NewParts*;

is the same as

Cost <− *Cost* **ujoin** *NewParts*;

or, for adding in place, which the **update** of course does,

Cost <+ *NewParts*;

We can assume *NewParts* is defined on the same attributes as *Cost* and has some tuples that are not in *Cost*.

update *Cost* **delete** *OldParts*;

is defined to be the same as

Cost <− *Cost* **djoin** *OldParts*;

except that the former deletes in place while the latter must copy the relation.

• • •

A comparison with SQL of the relational algebra described in this long section highlights two failures of SQL. The first is SQL's tendency to slip into thinking of individual tuples rather than whole relations, which happens particularly with updating. The second is SQL's failure to provide an explicit, infixed notation for operators, which we will see in the next section makes impossible the subsumption of the relational algebra into the domain algebra needed to deal effectively with nested relations. Finally, the QT-selectors, the variety of joins provided in Aldat's two families of binary operators, at least, and recursive views all transcend SQL capabilities.

6. Subsuming relational algebra into domain algebra: nesting

We have so far limited ourselves to first-normal-form relations, although the narrative has in places shown impatience with this restriction. It is a goal of programming language design that there shall be no second-class citizens among the data types of the language. Aldat has only relations and scalars. Relations, if they are constrained to first normal form, are second class: no relation may have relations as values of its attributes. So we break away from "simple" attribute values (scalars such as reals, integers, booleans and strings) and allow relations to nest within each other.

The literature on nested relations has introduced new language constructs and operators, particularly the nest and unnest operators. We would like to keep with our principles and attempt not to add new syntax. (Besides, nest and unnest have the peculiarly dissatisfying property that they are not inverses of each other, so we certainly do not want them as primitives.)

The idea is very simple. Since nesting admits relations among the attributes of relations, the domain algebra must admit the relational algebra among its operations. With this idea and almost no new syntax we can do everything we might wish, including nesting and unnesting.

We can show this by querying a nested version of *PartOf* (Section 2). (The attribute *numScrews* is virtual and will be defined shortly.)

nestedPartOf

(*A*	*SQ* (*S*	*Q*))	*numScrews* (*Q*)
`wallplug`	`cover`	`1`	
	`fixture`	`1`	
`cover`	`screw`	`2`	`2`
	`plate`	`1`	
`fixture`	`screw`	`2`	`2`
	`plug`	`2`	
`plug`	`connector`	`2`	
	`mould`	`1`	

NestedPartOf is a binary relation of four tuples. I have separated the tuples by horizontal lines to make this clear. The second attribute, *SQ*, is also a binary relation which happens to have two tuples in each occurrence in *nestedPartOf*.

Let's find the quantity, *Q*, of `screws`, one level down.

let *numScrews* **be** [*Q*] **where** *S*=`screw` **in** *SQ*;

I have written the virtual attribute, *numScrews*, outside the parentheses demarcating *nestedPartOf* above. Note that it is a nested relation. Two of its occurrences are empty and the other two are singletons. The definition of *numScrews* uses a (relational algebra) T-selector in the (domain algebra) **let**..**be** syntax. We now permit any declarative relational expression in the domain algebra.

If we actualize this virtual attribute, we still get a nested relation.

qtyScrews <− [*numScrews*] **in** *nestedPartOf*;

qtyScrews

(*numScrews*) (*Q*)
`2`

Note that we have only two tuples (separated by the line) because the two tuples of *nestedPartOf* with empty *numScrews* now become duplicates, since the projection has kept no other attribute to distinguish these tuples. Similarly, the two tuples of *nestedPartOf* with the same singleton value of *numScrews* also now are duplicates and one is elimintated in the projection.

The new virtual relations, *numScrews*, can be distinguished from each other if they include an attribute such as the value of *A* to do so.

let *A′* **be** *A*;
let *AnumScrews* **be** [*A′*, *Q*]
  where *S*=`screw` **in** *SQ*;

Here, *A*, which is a constant for any occurrence of *SQ*, has been included as an attribute of the virtual relation in exactly the same way as

let *One* **be** 1;

brought a constant, 1, into a flat relation in Section 4. This is a scoping mechanism. It supposes that attributes at the same level as or a higher level than some relational attribute are visible from that relation.

A′ will be created as an alias of *A* when *AnumScrews* is actualized:

AqtyScrews <−
  [*AnumScrews*] **in** *nestedPartOf*;

AqtyScrews	
(*AnumScrews*)	
(*A′*	*Q*)
cover	2
fixture	2

This result distinguishes the two nested relations that described quantities of screws, while still eliminating duplicate tuples with empty relations. It is still a nested relation.

To get a flat relation from this, we need to raise the level of the nested relation. We already learned how to raise levels from flat relations to scalars in Section 5.3. We need a singleton relation on one anonymous attribute. As before, this can be achieved with an anonymous reduction. But this time the reduction is on the relational attribute, not on a scalar.

AllScrews <− [**red ujoin of** *AnumScrews*] **in** *nestedPartOf*;

The result of this is the union of all the occurrences of *AnumScrews*. As a virtual attribute, this has the same value in each tuple, so when it is projected, duplicates are eliminated and only one union remains. Since the value is thus a singleton and the projected attribute is anonymous, the level is raised and we get a flat relation.

AllScrews	
(*A′*	*Q*)
cover	2
fixture	2

This is how unnest is achieved with no new syntax.

The selection condition may be extended to lower levels of a nested relation by using the boolean-valued nullary projection. Thus, to find assemblies that have screws one level down, we write

[*A*] **where** [ ] **where** *S*=`screw` **in** *SQ*
  in *nestedPartOf*;

I have now shown relational algebra operators in both scalar and aggregation domain algebra. The richness of the relational algebra allows immense flexibility for processing nested relations. Recall only that operators used with **red** and **equiv** must be associative and commutative: **ujoin**, **ijoin** and **sjoin** are, and Aldat has explicit operator names for them which can now be used after **red** and **equiv**.

To nest a flat relation, we need syntax to group attributes. We introduce the **relation**() operator into the domain algebra. This creates a singleton nested relation from the attributes that are its operands. The rest of nest just uses relational operators in the domain algebra. Here is the creation of *nestedPartOf* from *PartOf* (Section 2).

let *SQsingleton* **be relation**(*S*, *Q*);
let *SQ* **be equiv ujoin of** *SQsingleton* **by** *A*;
nestedPartOf <− [*A*, *SQ*] **in** *PartOf*;

Nesting makes possible another useful operator of the domain algebra, **transpose**. This can convert any relation into attribute-value pairs, or attribute-type-value triplets, or just list, as metadata, the attributes of a relation. Here is the full **transpose**, applied to the *Cost* relation.

let *xpose* **be transpose**(*attr, type, val*);

Cost				
(*Part*	*C*)	*xpose*		
		(*attr*	*type*	*val*)
wallplug	0.04	Part	strg	wallplug
		C	real	0.04
cover	0.10	Part	strg	cover
		C	real	0.10
fixture	0.03	Part	strg	fixture
		C	real	0.03
plate	0.06	Part	strg	plate
		C	real	0.06
screw	0.05	Part	strg	screw
		C	real	0.05
plug	0.01	Part	strg	plug
		C	real	0.01
connector	0.02	Part	strg	connector
		C	real	0.02
mould	0.08	Part	strg	mould
		C	real	0.08

The operands of **transpose** are metadata attributes (*attr, type*) or an attribute, *val*, of **universal** type. Data in such an attribute must store its type along with its value. (This is also true of data stored in union types, which I do not discuss here.) Like **grep** (Section 5.2), **transpose** identifies its operands not by position or name but by type. Thus for example *attr* is of type **attrib**, *type* is of type **type** and *val* is of type **universal**.

Metadata, and a slightly different form of **transpose**(), was introduced into Aldat in 2001 [55]. I will mention here only that metadata, **attrib** metadata in particular, demands two special operators. The **quote** operator is needed to allow an attribute name to be used without being dereferenced, say in a domain algebra comparison. The **eval** operator is needed to force dereferencing when the syntax would otherwise use the attribute name directly. Here is an example of the latter which serves to invert the **transpose** operator.

let eval((**attrib**)*attr*) **be** (*type*)*val*;

This translates into two statements for each tuple of *Cost*, above. For the first tuple, these are

let(*Part*) **be** (**strg**)`wallplug`;

let(*C*) **be** (**real**)`0.04`;

From this, *Cost* can be regained from *xpose*, the details depending on just how far the transpose operation was originally taken.

We have used transpose for datacube analysis and classification data mining [55,62], and for schema discovery in semistructured data [57]. Recent work [84] has independently formalized rather more metadata operators than **transpose**, **eval** and **quote**, and I leave reducing those operators to the Aldat ones as an exercise. (Their "transpose" is actually the inverse of Aldat's; both approaches have probably misused the word, since neither "transpose" is its own inverse.)

Programming language experience nudges us one step further. Nested relations offer a data structure in which relations are first class. Why should a relation not be able to be nested in itself, creating a recursive data structure? LISP was built on such data structures. The bill of materials provides a good example (Table 6).

Note that the data determines the depth of recursion, and the $\mathcal{DC}$ null value terminates the paths.

To find all components from this, the domain algebra must allow recursive definitions of virtual attributes across levels of nesting.

let *cmpnt* **be** *component* **ujoin**
    [**red ujoin of** *cmpnt*]**in** *subassembly*;

This is true domain algebra recursion, unlike the self-referring domain algebra definition of Q in Section 5.4. It can be recognized by the recursion descending one nesting level by means of the anonymous projection. As with relational recursion, *cmpnt* is initially empty. At the lowest level, it then takes on the value of *component*. The result of

AllComponents <− [*cmpnt*] **in** *assembly*;

is

AllComponents
(*cmpnt*)
`wallplug`
`cover`
`fixture`
`plate`
`screw`
`plug`
`mould`
`connector`

Instead of imposing recursion on the programmer, we could follow Cruz, Mendelzon and Wood [19] and provide *path expressions*. The path expression equivalent to the above includes a Kleene star.

let *cmpnt* **be** (*subassembly*/)**component*;

and we could avoid even the domain algebra by writing

AllComponents <−
    assembly/(*subassembly*/)**component*;

—which produces *AllComponents*(*component*) instead of *AllComponents*(*cmpnt*).

The recursive domain algebra is strictly more powerful than the path expressions, and the latter may be considered syntactic sugar defined by the former. (Caveat: I have drastically simplified the history of path expressions by the one citation. Path expressions evolved from the notation in G+ [19], which is in fact more expressive than its successors. The notation used here is the recent notation used for semistructured data, about which more presently.) An example of a problem which is beyond the syntactic sugar but expressible in recursive domain algebra is a program to create a relation describing all the paths

assembly								
(*component*	*subassembly*							)
	(*qty*	*component*	*subassembly*					)
			(*qty*	*component*	*subassembly*			)
					(*qty*	*component*	*subassembly*	)
`wallplug`	1	`cover`	1	`plate`	$\mathcal{DC}$			
			2	`screw`	$\mathcal{DC}$			
	1	`fixture`	2	`plug`	1	`mould`	$\mathcal{DC}$	
					2	`connector`	$\mathcal{DC}$	
			2	`screw`	$\mathcal{DC}$			

Table 6: recursively nested bill of materials.

in a nested relation [57,60]. This uses **transpose** recursively.

Other useful notation is a “wildcard” symbol, which can stand for any attribute name in a path, and which allows recursive domain algebra to be applied to even nonrecursively nested relations. Path expression symbols for alternatives and options are also important [57,60].

A comparison of processing the flat bill of materials, *PartOf*, with processing the nested *assembly* reveals that the flat version is very good for breadth-first traversal of the bill-of-materials hierarchy. The recursively nested version, on the other hand, favours depth-first traversal. Programmers and language implementors should bear this complementarity in mind.

The premier application of recursive nesting and recursive domain algebra is semistructured data [59,57]. The family tree example in Section 2 is a beginning and can now be understood in light of the above discussion. Of course, semistructured data needs more than recursive data structures and the means to process them. Flexible typing, as captured by union types, polymorphism, and pattern-matching operators such as **grep** (Section 5.2), are all also needed. To convert from marked-up text to a (recursively) nested relation requires a special operator which, like **transpose** is not recursive itself but restricted to a single level of nesting, but which, through the recursive domain algebra, can be applied to all levels. **Transpose** may be used recursively for schema discovery, producing a nested metadata relation containing the schema of some other relation as discovered by traversing the contents of that relation. I say more about semistructured data in Section 8.

A reason for the early restriction of relations to first normal form is the difficulty of seeing how to implement non-first-normal-form relations, especially on secondary storage. Aldat does this by representing them as flat relations. Indeed, nesting, and even recursive nesting, adds nothing to the functionality of Aldat, except by providing a different point of view, which we have found liberating in a number of ways.

I will not show in this paper the flat-relation implementation of nesting, but it is important to say that it permits not only the tree-structured hierarchies I have discussed so far in this section but also hierarchies which are DAGs (directed acyclic graphs) and even non-hierarchies with cycles. Aldat allows nested relations to exploit this with simple mechanisms, involving union types, to indicate (hierarchical) common subexpressions and (non-hierarchical) crossreferences [57].

7. Integrating procedural abstraction and relations: computation

We have now taken purely relational formalism just about as far as I think that it can go. I have not yet touched on programming language constructs. What we have so far is the arithmetic of relations. A programming language for numbers goes much further than just offering arithmetic. That is the difference between a programming language and a calculator. Similarly, a programming language for relations should provide more than relational and attribute operators, even though we have seen that these can be quite sophisticated.

The feature among all others that most distinguishes a programming language is procedural abstraction. The simplest form of procedural abstraction is the function call of classical programming languages. Although such functions are not always mathematical functions (many-to-one mappings between sets), the name is suggestive. A mathematical function is a special case of a mathematical (binary) relation, and database relations generalize this. Why could a programming language function not be a special case of a programming language relation? The extension that would be needed would be to a form of procedural abstraction whose parameters could be treated at different times as inputs or as outputs, and not be frozen exclusively into input or output. Thus, the relationship implied by $sum(a, b, c)$ could variously be used to find c as the sum of a and b, b as the difference of a from c, or a as the difference of b from c.

As well as implementing a relationship among the parameters instead of just a prescription with fixed input and output, this approach can lead to an abstraction which is just a relation. Thus it can be invoked by existing syntax of the relational algebra: T-selectors and joins, for instance.

To define such a relationship, we need a mechanism to provide alternative blocks of code to be executed depending on which parameters are inputs and which are outputs in any given invocation. (We do not attempt to generate the right code automatically from the relationship: that is still open territory in the field of constraint programming languages.) So a little new syntax is required. While I am adding syntax, I will give the name "computation" to our generalization of functions, and use the abbreviation **comp**. (This can also be understood to abbreviate "compressed relation", since, as we shall see, the **comp** often describes a relation with an infinite number of tuples.)

Here is the definition for the sum.

```
comp sum(a, b, c) is
{ c <−a + b } alt
{ b <−c − a } alt
{ a <−c − b };
```

Clearly, the programmer is responsible for correctly expressing all relevant aspects of the relationship being considered. The language implementation will not object if $b <\!\!-c/a$ were specified instead of the difference, but the results may be confusing. Note that a, b and c are attributes and must be declared beforehand (in this case as integer or real numbers): they are not parameters in the sense that is conventional for procedures and functions.

Invoking computations requires no new syntax. For example, we can use a T-selector to add 2 and 3.

Sum23 <−[c] **where** $a = 2$ **and** $b = 3$ **in** *sum*;
Alternatively, in the context of a relation, $R(a, b)$, *sum* may be invoked by a natural join and perform multiple additions.

```
relation R(a, b) <−
   {(2, 3), (1, 4), (−2, 7)};
SumR <− sum ijoin R;
```

giving (I show R as well)

```
R(a   b)   SumR(a   b   c)
  2   3          2   3   5
  1   4          1   4   5
 −2   7         −2   7   5
```

(I have, in Section 5.2, already shown syntactic sugar for the very special kind of T-selector we have just used. Computation invocation is not excluded. We can write the above as *sum*[2,3], and the differences, respectively, as *sum*[2,,5] and *sum*[,3,5]. This makes computation invocation look like the familiar function call.)

The computation can also be used at top level, for instance in this case on the top-level scalars, A and B. Since the invocation is not in the context of relational algebra, we need syntax to specify the inputs and the output,

sum(**in** A, **in** B, **out** C);
will produce a top-level scalar, C, for the sum of the values in A and B.

A top-level invocation is probably more interesting when the computation is defined on relations.

```
comp decompose(R, S, T, A, B, C) is
{ R <−[A, B] in T;
  S <−[B, C] in T} alt
```

```
    { $T$ <−$R$ **ijoin** $S$ };
```

Here is the invocation that does the projections.

```
  decompose(out R,out S,in T, in A,in B,in C);
```

Here is the invocation that does the natural join.

```
    decompose(in R,in S,out T, in A,in B,in C);
```

Note that the attributes, which are needed in the first case, are **in**puts to the computation. They are **in**puts to the second invocation, even though they are unused: an implementation could get confused by failing to find an **alt** block which creates them if they had been called **out**puts.

Computations may be recursive

```
  comp gcd(k, m, g) is
  { g <− if k < 0 then gcd[−k, m] else
          if m < k then gcd[m, k] else
          if k = 0 then m else
             gcd[m mod k, k]
  };
```

but it is a challenge to devise an application requiring recursion on more than one **alt** block. (The square brackets in *gcd*$[m, k]$ are the special syntax for T-selectors discussed above and in Section 5.2. Function invocation in computations is identical to array lookup in relations.)

Two particular adaptations of procedural abstraction in general, and hence computations in particular, are event programming and object orientation.

Event programming. Specially named computations form event handlers. Since an event, in the context of event programming, can be defined as an invocation of a parameterless procedure by the system (as opposed to by the programmer), an event handler is exactly a procedure. Since, for completeness, event handlers should be able to initiate new updates, and even updates that recursively invoke the event handler itself, I can illustrate the idea briefly with a quite unusual event handler.

```
  domain n intg;
  relation start(n) <− {(4)};
  comp post:add:iota() is
  { let nmin1 be (red min of n) − 1;
    let n be nmin1;
    update iota add
       [n] in [nmin1] where nmin1>= 0 in iota;
  };
```

This event handler is invoked after the system updates *iota* by adding data. So it invokes itself. Here is the invocation that starts the whole process.

```
   update iota add start;
```

What this example does is generate *iota*(n) containing five tuples with n having values `0` to `4`: once *start* has added the tuple (4) to *iota*, is to invoke the event handler to add (3), which invokes it again to add (2), then a third time to add (1), a fourth time to add (0), and a final time to discover that *nmin1* is negative and perform no update.

It is understood that a void update, which adds nothing, stops the process.

The event handler is a conventional computation but with a constructed name which gives the relation, whose update invokes the handler, the nature of the update (**add**, **delete** or **change**), and whether the handler is invoked after (**post**) or before (**pre**) the update is done. There is a mechanism, not discussed here, to remember the prior state or anticipate the updated state, respectively. (This mechanism also can detect which parts of the relation are updated, an "update count" which avoids thinking of tuples.)

Object orientation is an important programming language facility which has been abused by misunderstanding (largely due to the unfortunate name [37]) and, to some extent, by overattention. We find that the significant contribution made by object orientation is its mechanism for instantiation—creating "objects" from classes. Instantiation can be done by older programming languages, but at the expense of making and maintaining some data structure, such as an array, to manage all the instances; object-oriented languages automate this work. What is instantiated, of course, is the *state*, and so state is fundamental to object orientation. This state is encapsulated in an entity which contains both code and the state, but the code is "re-entrant", i.e., can be shared by many invocations, and so does not need to exist in more than one instance. The entity with instantiated state is called an "object", but it is essentially an abstract data type (ADT) with internal variables.

Confining state to its ADT is a way of limiting the damage it may cause, simply because the values of the variables representing the state are not up-front and visible ("referentially transparent") in a program, and the programmer must remember what the meanings of the state are at different points in the program. A whole discipline, "declarative" or "functional" programming, has developed ways of avoiding state almost completely, but at the burden of carrying around explicitly the information otherwise contained in the state. Object orientation is the compromise that accepts state but limits it.

Conventional object orientation requires syntax for describing classes, and for indicating which variables and procedures in the class are hidden from users of the class, and which are publically accessible. It also requires syntax to instantiate the class as objects, usually one object at a time using a command, **new**. Following [3] we can avoid all new syntax for declarations, apart from syntax to declare a **state**. Using relational ideas we can avoid introducing a **new** command and also avoid limiting instantiation to only one object at a time.

Before we look at this adaptation of computations and relations, I briefly discuss other aspects of object orientation which are commonly considered significant. The first is polymorphism. This is another important programming language facility, but all I need say is that polymorphism is independent of object orientation: there can be polymorphic languages which are not object-oriented, and there can be languages with mechanisms for instantiation which are not polymorphic.

The second is inheritance. This arises from the collective nature of the instantiated objects. Since they form sets, or classes, it is legitimate to think of subclasses: these not only consist of a subset of the instances of the parent class, but they usually also have additional, specialized behaviour, reflected in the code encapsulated in the subclass. Importantly, subclasses share behaviour with parent classes: this behaviour is said to be inherited, and inheritance mechanisms save rewriting the code for the shared behaviour. The ability to re-use code is touted as a significant benefit of object-orientation, but it really is an admission of defeat in a major goal of programming language design, which is to provide facilities so simple and powerful that it is easier to write a program from scratch than to figure out what somebody else did. Relations, which are sets, can also contain or extend each other. I will not here embark on a description of these possibilities [67].

The third aspect of object orientation is "complex objects". This especially arises in connection with object-oriented databases, and is the basis for repeated assertions that object-oriented databases must replace relational databases because relations cannot represent complex objects. Our discussion of nested relations in Section 6 removes the force of these claims. Even prior to nested relations, the Aldat mechanisms for operating on aggregates of tuples (the aggregation operations of the domain algebra, the σ-joins, and the QT-selectors) provide much of the functionality, not found in commercial relational database systems, that these criticisms say is missing.

I return to specifying "classes" and instantiating "objects" with minimal new syntax. The central concept is state, which in this context means simply the presence of variables which are private to the entity (class or procedure) and which retain their values between invocations. Mechanisms for this predate object orientation. For example, Algol 60's **own** variables were introduced to do it. In recent languages, such variables are called **static**, an approximation to the clearer term, **state**. Accordingly, I introduce a **state** declaration into computations.

The simplest example of an "object" is a counter, so I implement one with a computation.

```
comp counter(ct) is
   state count intg;
   { count <- ct
   } alt
   { count <- count + 1;
     ct <- count
   };
```

Counter has two **alt**-blocks so that we can use one invocation to initialize it,

```
counter(in 0);
```

and the other invocation to count,

```
counter(out ct);
```

(*ct* now is 1)

```
    counter(out ct);
```

(*ct* now is 2), and so on.

The *counter* example shows state, but not instantiation. We move on to a bank account example in which one computation contains, and outputs, other computations: computations are "first class" data types, like relations. The outer computation serves as the class and has the state (the bank balance). The contained computations serve as the methods (to deposit/withdraw, and to determine the balance).

```
    domain DEP, BAL intg;
    domain DEPOSIT comp(DEP);
    domain BALANCE comp(BAL);
    comp ba(BALANCE, DEPOSIT) is
       state bal intg;
       state oldbal intg;
       { comp DEPOSIT(DEP) is
          { oldbal <− bal;
             bal <− bal + DEP}
          alt
          { DEP <− bal − oldbal};
          comp BALANCE(BAL) is
          { BAL <− bal};
          bal <− 0;
       };
```

The *ba* "class" does three things when invoked. It defines and makes available through its attribute list the two "methods", *BALANCE* and *DEPOSIT*. Third, it initializes one of its **state** variables, *bal*, using the statement *bal* <− 0;. I will explain and illustrate the methods shortly, but first we must instantiate as many bank accounts and balances as we need. Here is a relation with two accounts.

```
    relation accts(ACCNO, CLIENT) <−
       { (1729,"Pat"),(4104,"Jan") };
```

Instantiation is done by invoking *ba* over the relation *accts*: in the usual way, by **ijoin**.

```
    accounts <− accts ijoin ba;
```

The result is a relation on all four attributes, two from *accts* and two from *ba*. Since none of these is shared, the result is a Cartesian product of two tuples (Table 7).

DEPOSIT and *BALANCE* are constant attributes, making the (re-entrant) code of the computations available to each tuple. I have also shown one of the state variables, *bal*, in the attribute list, but in braces to emphasize that it is not an attribute with a name which is known outside of the *ba* class: it is a hidden state. It has the value 0 in each tuple because the invocation of *ba* by the join executed the statement initializing *bal*.

BALANCE can be invoked by the domain algebra, because it changes nothing in any tuple.

```
    let balance be BALANCE[ ];
    balance where ACCNO = 4104 in accounts;
```

So can the lookup mode of *DEPOSIT*, which uses the second state variable, *oldbal*, to find the last deposit made:

```
    let last_deposit be DEPOSIT[ ];
    last_deposit where ACCNO = 4104 in
       accounts;
```

On the other hand, the first mode of *DEPOSIT* changes *bal* and *oldbal*, and so must be used in an update.

```
    update accounts change DEPOSIT(in 100)
     using where ACCNO = 4104 in accounts;
```

What this accomplishes is *hiding*, the central goal of abstract data typing, and encapsulation, the central goal of object-orientation. The stored tuples of a relation themselves can always provide an alternative to the state in object-orientation, but do not permit data such as somebody's bank balance to be concealed. The above code provides true data abstraction by requiring access to the hidden state only through the "methods" associated with the class.

Object-oriented databases have an immense literature, and form only a subset of object orientation in general. They were introduced to bring to databases the benefits of encapsulation and data hiding that object-oriented programming languages (Smalltalk at first) provide, and were seen as irreconcilable with the openness of relational data. There was also a performance benefit for the special cases in which the relevant part of a relational join can be implemented (much faster) by pointer dereferencing. The above discussion of object orientation in Aldat reconciles the apparently irreconcilable and captures all the linguistic benefits of object orientation. (I have not discussed pointer-dereferencing implementation, but this can be either an automatic option of the im-

accounts

(*ACCNO*	*CLIENT*	*DEPOSIT*	*BALANCE*	[*bal*])
`1729`	`Pat`	<*DEPOSIT* code>	<*BALANCE* code>	0
`4104`	`Jan`	<*DEPOSIT* code>	<*BALANCE* code>	0

Table 7: relation for instantiating bank accounts.

plementation, based on selectivity predictions, or built in to syntax, which I have not given here, for inheritance.) Aldat is not an "object-relational" compromise, but fully integrates relations and object orientation, without "impedance mismatch", by exploiting the origins and essentials of object orientation.

8. Relations and the Internet

Two aspects of internet databases serve to illustrate some of the constructs we have developed above. The first is data on the World Wide Web, which becomes an exercise in semistructured data, and also illustrates the philosophy adopted by relational query languages and by Aldat that releases the user or programmer from needing to distinguish between data in primary memory or on secondary storage. The second introduces an internet protocol which can serve as a low-level basis for distributed database programming.

Names discussed hitherto in this paper have all been database-local, although no distinction has been made between primary memory and secondary storage. In this section, I break that restriction by extending to remote names. Eventually, no distinction should be made between local and global names.

8.1. Data on the Web

Data on the World Wide Web is formatted in the GML family of markup syntax, mostly HTML or XML. The family tree data from Section 2 becomes an example which uses many important aspects in HTML: tags, attributes, interior references, local references and embedded images.

```
<HTML> <!file famtree.html>
<Head><Title>Family Tree</Title>
</Head><Body>
<A href=bioTed.html>Ted</A>
<A href=#TedAliceWedding>married</A>
Alice in 1932. Their children,
Mary (1934) married Alex in 1954
(Joe was born to Mary and Alex in
1956) and James (1935) married
Jane in 1960 (James and Jane had
Tom in 1961 and Sue in 1962).
<A Name=TedAliceWedding> <br>
<img src=Ted_Alice.jpg width=400><br>
Ted and Alice Just Married, 1932 </A>
</Body>
</HTML>

<HTML> <!file bioTed.html> <Head>
<Title>Ted's Biography</Title>
</Head> <Body>
Ted was born at McGill University
and worked there until retiring.
</Body>
</HTML>
```

The relational representation of this data uses recursive nesting to include in the web page every other web page picked up by a URL within an anchor tag, <A..>. This frees the programmer from concern about page fetches and treats the whole Web as if it were internal to the client host: database languages typically avoid explicit accesses to secondary storage, and Aldat extends this automation to the Internet.

domain
  A(*content, href, Head, Body, Name, img*);
domain *img*(*src, width*);
domain *BodyRel*(*content, A, img*);
domain *Body* **strg**|*BodyRel*;
domain *Head*(*Title*);
relation *HTML*(*Head, Body*);

(Instead of making all these declarations manually, we could depend on a special operator to generate the recursively nested relation directly from the marked-up text.

mu2nest(*content*):
This operator creates an attribute, *content*, for text in the HTML that is not delimited by tag and endtag symbols. It is not intrinsically recursive and so must be applied by recursive use of the domain algebra. I do not elaborate here. See [57].)

Since the resulting nested relation is too wide for this page, I explain the above declarations in words. The Web is captured in the relation *HTML*, which has *Head* and *Body* as attributes. *Head* consists (for this example) only of the string, *Title*. *Body* may be just a string, or alternatively it may include, as well as the string (*content*), images (*img*) and anchors (*A*): thus it is defined as a union of string or *BodyRel*. An image has attributes *src*, giving its filename, and *width*: note that these "attributes" in the HTML sense become attributes in the relational sense, on a par with the attributes that represent components in the form of scalars or nested relations. Finally, anchors, *A*, recursively contain further pages, and so each has a *Body*. It will also have a *Head* (e.g., with *Title* `Ted's Biography` in the case of the anchor with URL `bioTed.html`). Alternatively, *A* may have a *Name*, if it is the label for an interior reference. In the example, the *A* with *Name* `TedAliceWedding` also has *img* as an attribute. The URL used by the anchor is the value of the *href* attribute (in both senses) of *A*. Finally, *A* itself has *content*, such as `Ted` for the first *A* of the example or `married` for the second.

An example query on this structure would be to find all *href*s from this document:

HTML(/.)**href*;

The answer is {`bioTed.html, #TedAliceWedding`}. (But be careful! The answer could be very large).

The domain algebra and **grep** help with one problem, raised for instance in [46]: can we, for the sake of retrieval speed, express the distinctions among interior, local and global URLs? An interior URL refers to another part of the same document, and syntactically begins with a #. A local URL is in the same directory as, or a descendant directory of, the one containing the referring document, and syntactically does not begin with a /. (These syntactic specifications are simplistic, to avoid longer discussion: comparisons of the cited *href* with the citing *href* can also be made.) A global URL is anything else.

let *interior* **be grep** `"^#"` **in** *A*;
let *local* **be grep** `"^[^/]"` **in** *A*;
let *global* **be where**
((**not** [ ] **in** *interior*) **and not** [ ] **in** *local*)
in *A*;

If we wanted to pursue only interior URLs, we could use the path expression

HTML/*Body*/*interior*/*href*

Note that virtual attributes are just as acceptable to path expressions as are actual attributes.

The facilities described in this section and in Section 6 are elaborated in [59] and in [57], where they are compared with the classical work on semistructured data. (A single citation [1] captures most of this literature). To my knowledge, the capabilities of XML and XQuery [9] are also contained in Aldat, with negligible extension. This is not just a relational approximation to, or partial implementation of, these languages.

8.2. Distributed database computing: the Aldat protocol for names

The hierarchy of relations found in nesting resembles the hierarchy of files and directories in a file system. We could possibly use the path expression syntax to navigate among databases contained in each other the way directories are, or among databases merely situated in different parts of a directory hierarchy. Taking this further, we could even bridge different hosts by path expressions navigating such a hierarchy [7]. Following the hypertext transfer protocol, HTTP, we devised a protocol to connect various Aldat databases, prefixing the path expressions with its name, *aldatp*.

Figure 8.2 shows a collection of databases (underlined names, such as *public_aldatp, pubA, privB*) containing entities, *E1 .. E8*, which may be relations, scalars or computations. (F3, F4, F7 and *F8* are to be created by code which I will discuss shortly.)

All we need to do to write code that refers to entities in other directories, even on other hosts, is to extend the name of the entity to a full path expression prefixed with the protocol. Thus, sup-

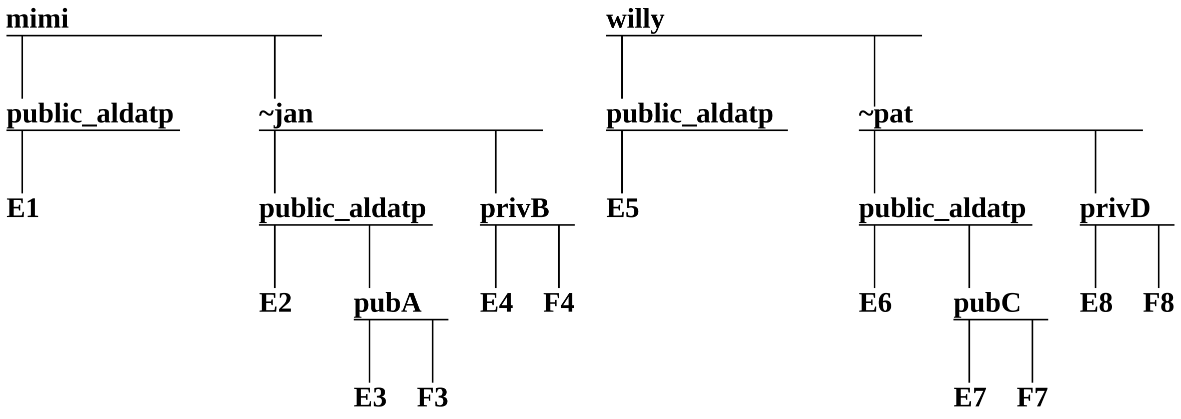

Figure 8.2: a distributed database.

pose we are running in Jan's private database, *privB*, and want to create a copy of *E3* in Jan's public database *pubA* in *public_aldatp*.

F4 <− *aldatp://mimi/∼jan/pubA/E3*;

There is no reason why extended names cannot be used on the left-hand side of an assignment. If we are running directly in Jan's *public_aldatp*, and assuming we have permission, we will copy *E2* to *F3* in the contained database, *pubA*.

aldatp://mimi/∼jan/pubA/F3 <− *E2*;

These are both multidatabase examples: they are confined to the one host, *mimi*. We could equally well reach out from *mimi* to create an entity on *willy*.

aldatp://willy/∼pat/privD/F8 <− *E2*;

(Presumably, Pat has set permissions for us to do this, or has given us a password.)

Not only entities but also statements and expressions can be qualified by an aldatp path expression. Here, someone running anywhere creates a copy of Pat's *E7* in the same database, using a remote statement.

aldatp://willy/∼pat/pubC/{*F7* <− *E7*};

And here, Pat, running in *pubC*, does a semijoin of relation *E7*(B, C) with *E3*(A, B), using a remote expression.

```
(aldatp://mimi/∼jan/pubA/(E3 natjoin
    aldatp://willy/∼pat/pubC/([B] in E7)))
    natjoin E7
```

Other posibilities resemble these examples. Any name anywhere may be qualified at any point in Aldat code. Note that we must use full aldatp path expressions to avoid syntactic confusion with path expressions for nested relations. The concepts are so similar that we decided to share syntax, and perhaps they will someday be unified to the same concept.

9. Advanced implementation techniques

Implementation of the research system, *relix*, which has tested the language constructs described in this paper, has been simplistic, using sequential files, sorting and merging. This has kept the implementation of new ideas within the scope of student theses and projects [36,87], [10,88,32,81,12], [41,21], [13,42], [91,11,34,30,23, 22], [86,33], [4,43,79], [78,38], [90,67], [31,75], [29], [85,2], [83].

In this section, we look at two data structures developed in the course of the Aldat project which are more advanced than sequential files and which might form bases for more professional implementations. Multidimensional paging, or *multipaging* [49,63,73] and tries [64,70–72], [66,77,89,25,61] both support the multidimensional, dynamic data needed by relations, which are time-varying and must be accessible on any attribute.

Multipaging is a direct-access structure which preserves order in multiple dimensions simultaneously, and so is suitable for range searches and for partial match queries along any of its axes. The static version [49] is a precursor to the grid file [69] and has the advantage of requiring only negligibly more space than the original file. At the cost of sorting the data along each dimension, it uses a Viterbi-like heuristic [82] to discover if the data distribution is amenable to allocating essentially equal numbers of records to each of a rectilinear arrangement of pages. The dynamic version [63] solves the problem of how to make this rectilinear arrangement dynamic: essentially how to represent dynamic multidimensional arrays [39].

Tries support sublinear substring and pattern matching in text, and their advent [20,24] predates the earliest linear pattern matching algorithms [40,6]. Their special use for text [68] indexes all substrings and is tantamount to indexing N **choose** 2 characters for an N-character text. Since tries store common prefixes only once, they can compress such indexes enormously [89]. Orenstein [72] discovered pointerless representations requiring not more than 2 bits per node, and Shang [65] used them to reduce such index sizes to less than $3N$. Independent work on suffix arrays [26,45] has not yet exploited this property of tries—see, e.g., [28]. Representing relations requires multidimensional tries, or kd-tries, easily adapted from kd-trees [5]. These induce a one-dimensional ordering of d-dimensional space, which, for $d = 2$, is the first known fractal [74].

10. Conclusion

The goal of Aldat is to exploit fully the relational model of data and the algebras of operators on relations and on attributes. Relations and the relational and domain algebras were conceived for secondary storage because their level of abstraction is suited to the bulk processing of data dictated by the high latency and other delays intrinsic to secondary storage. Latency is also a property of computer networks and the Internet in particular, so the abstractions in Aldat make it suitable for internet programming as well as secondary storage programming. High abstraction is good in general, and programming in Aldat is at a higher level than the vast majority of programming languages, so we are motivated to make Aldat a general-purpose programming language. In particular, the abstraction over looping embraced by the relational and domain algebras means that programmers need not write loops when the order of execution is irrelevant: a result is that parallel programming is intrinsic to Aldat.

The domain algebra is an innovation in Aldat. By its independence from relations it provides important intellectual simplification through separation of concerns. Operators in the domain algebra include scalar operations on simple attributes, but by subsuming the relational algebra they also provide the functionality needed for nested relations. Taking nesting all the way makes relations a recursive data structure, and operations on this are supported by recursion in the domain algebra. An application is semistructured data, and specializing the recursive operations leads to the syntactic sugar of path expressions.

Aldat takes a systematic approach to the classical relational algebra, dividing the binary operators into two families which it extends fully, and fusing the unary operators into a single one which it extends to quantification. Special, whole-relation operators are provided for regular expression matching. A family of editors is introduced to interface between the Aldat programmer and the end-user. Views support the relational recursion needed for transitive closure operations (and, incidentally, for inference engines). Updates are treated systematically, avoid tuple-at-a-time processing (which violates the relational level of abstraction), and, in combination with the foregoing algebraic operators, are very powerful. A synchronization mechanism for concurrent programming is an adaptation of the basic unary operator of the relational algebra.

Procedural abstraction takes Aldat beyond data structures and operators. Even this mechanism, so essential to programming languages, has relational aspects, and Aldat generalizes procedures to a mechanism with multiple alternative choices of inputs and outputs among its parameters. This *computation* allows Aldat to support a variety of advanced constructs including event handlers and abstract data types with state (i.e., object classes).

Another central programming language mechanism is scoping. Aldat considers a scope to be equivalent to a database and to a tuple, and we show the connection in practice by treating multidatabases and databases distributed over the Internet within the Aldat framework. The ideas, and therefore the syntax, also tie in with nested relations and the path expressions of recursive nesting and semistructured data.

As the title makes clear, Aldat is not yet complete. Two outstanding tasks are to conceive and build a fully integrated transaction mechanism, and to design and build a trie-based implementation for the whole system. A number of other items for finishing up are mentioned or implied in the text.

This paper is intended to be a fairly self-contained overview of the Aldat language and its actual and potential implementation. Space limits us from discussing many details. These may be found in the papers, theses and projects cited, and on the Web [54].

11. Acknowledgements

While this paper is a retrospective on the thinking of a single principal investigator, the work was all done by dozens of students over the years, many of whom are not explicitly cited in the text. Those mentioned built implementations to test the ideas and integrate them into an ongoing system. Others did preliminary explorations and followed leads that subsequently turned out to be less productive than the directions discussed here, but their work has been just as important to the Aldat project.

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. The Québec and Canadian taxpayers have covered salaries, grants and scholarships.

I am grateful to Dennis Shasha for the invitation to write this paper for *Information Systems* and for his careful editing. I thank Bipin Desai for having given a precursor of the paper a special session at the *IDEAS* conference held in Montreal in July 2005.

Several people have read the paper thoroughly from the perspectives of databases, programming languages and computer scientists-in-training. I am indebted to Shamir Biswas, Bettina Kemme, Jack Orenstein, John Plaice, Simon Raboczi, Andrew Schaaf and Peter Schachte. Remaining errors and stubbornness are my own.

The paper is dedicated to the memory of Edgar Frank Codd, on whose work it is entirely based.

A. The motivating examples

This appendix is for readers who wish to follow the details of the five examples shown in Section 2.

1. The matrix multiplication code,

let *ab* **be equiv** + **of** $a * b$ **by** i, k;
AB <−[i, k, ab] **in** (A **natjoin** B);

can be illustrated with integer 2×2 matrices.

Inputs				Output		Input relations					
A		B		AB		$A(i$	j	$a)$	$B(j$	k	$b)$
3	4	5	0	43	24	1	1	3	1	1	5
0	5	7	6	35	30	1	2	4	2	1	7
						2	2	5	2	2	6

The natural join is done first. We visualize the virtual attribute, *ab*, and also the unnamed expression, $a * b$, that *ab* is derived from.

A **natjoin** B

$(i$	j	k	a	$b)$	$a * b$	ab
1	1	1	3	5	15	43
1	2	1	4	7	28	43
1	2	2	4	6	24	24
2	2	1	5	7	35	35
2	2	2	5	6	30	30

Finally the projection is done on *ab* and the index attributes, i and k, and I leave this to the reader to write out.

2. The inference engine code,

NewFacts **is** *Facts* **ujoin** [*Concl*] **in**
  (*NewFacts*[*Concl*: **sup** :*Ante*]*Horn*);

is equivalent to the loop

```
until NewFacts no longer changes
{ NewFacts <− Facts ujoin [Concl] in
    (NewFacts[Concl: sup :Ante]Horn);
}
```

with *NewFacts* initially set to empty.

As initial data, let's take

Facts(	*Concl*)
	`has stamp`
	`has envelope`
	`has enclosure`
	`enclosure is paper`

and

Horn

(*Rule#*	*Ante*	*Concl*)
1	`has enclosure`	`is postable`
1	`has envelope`	`is postable`
1	`has stamp`	`is postable`
2	`is postable`	`is letter`
2	`enclosure is paper`	`is letter`
3	`is postable`	`is greeting`
3	`enclosure is card`	`is greeting`

After the first iteration, *NewFacts* is the same as *Facts*, since the initially empty *NewFacts* produces an empty join with *Horn*.

In the second iteration, the **sup**erset σ-join produces the *Concl*usion, `is postable`, by firing *Rule#* `1`, since all its *Ante*cedents are now found in *NewFacts*, and the **ujoin** adds this conclusion to the tuples of *NewFacts*.

In the third iteration, *NewFacts* now has all the *Ante*cedents for *Rule#* `2`, so **sup** fires that, producing *Concl*usion, `is letter`, to be added to *NewFacts*.

No new rule is fired in the fourth iteration, because **sup** cannot find in *NewFacts* all the antecedents of the only remaining rule, 3, so *NewFacts* does not change and the iteration stops.

Inspecting this execution, the reader will see what I meant by "ultra-naive" implementation in Section 5.4, and should refer to that discussion for the sophisticate's angle.

3. The "explosion" of the bill of materials is discussed in Section 5.4, but the thinking is subtle and a worked example will help.

```
**let** $A'$ **be** $A$; **let** $S'$ **be** $S$; **let** $Q'$ **be** $Q$;
**let** $Q''$ **be equiv** $+$ **of** $Q * Q'$ **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** $:A, S']$ $[A, S', Q'']$ **in**
  (*Explo* $[S$: **natjoin** $:A']$ $[A', S', Q']$ **in**
  *PartOf*));
```

The recursive view is, as with *Horn*, implemented by replacing **is** with $<-$ and looping until *Explo* no longer changes, with *Explo* initially empty.

I will add a tuple to the *PartOf* given in the text, to illustrate the combination of paths of different lengths achieved by the sum $Q + Q''$. To compensate and keep the example short, I drop the two tuples for `plug`.

PartOf(*A*	*S*	*Q*)
`wallplug`	`cover`	`1`
`wallplug`	`fixture`	`1`
`cover`	`screw`	`2`
`cover`	`plate`	`1`
`fixture`	`screw`	`2`
`fixture`	`plug`	`2`
`wallplug`	`screw`	`3`

After the first iteration, *Explo* has the same tuples as *PartOf*, above, because of the empty **natjoin**.

The result of the **natjoin** with the renamed attributes of *PartOf* in the second iteration is

Explo$[S$:**natjoin**:$A][A', S', Q']$...

(*A*	$S \mid A'$	S'	Q	Q')	Q''
`wallplug`	`cover`	`screw`	`1`	`2`	`4`
`wallplug`	`fixture`	`screw`	`1`	`2`	`4`
`wallplug`	`cover`	`plate`	`1`	`1`	`1`
`wallplug`	`fixture`	`plug`	`1`	`2`	`2`

(Note that I have shown both aliases for the join attribute as $S \mid A'$, and the virtual Q'' is shown outside the parentheses.)

Still in the second iteration, the **ujoin** and the virtual Q''' give

PartOf $[A, S$: **ujoin** $:A, S']$ $[A, S', Q'']$...

(*A*	*S*	Q	Q'')	Q'''
`wallplug`	`cover`	`1`		`1`
`wallplug`	`fixture`	`1`		`1`
`cover`	`screw`	`2`		`2`
`cover`	`plate`	`1`		`1`
`fixture`	`screw`	`2`		`2`
`fixture`	`plug`	`2`		`2`
`wallplug`	`screw`	`3`	`4`	`7`
`wallplug`	`plate`		`1`	`1`
`wallplug`	`plug`		`2`	`2`
(*A*	*S*			*Q*)

where the very last line gives the (renamed) attributes that finally contribute to *Explo*.

The third iteration finds no change and the process stops.

Notes:

- A', S', Q' just rename A, S, Q (Section 4): this renaming is needed because, to avoid losing information, we use **natjoin** instead of **icomp** (Sections 5.1, 5.4).

- The apparently circular definition of Q''' is resolved by the projections used to actualize it (Section 5.4).

4. To explain the computation, *rotate*, I will just show all the possible invocations, using selections from the values $x = 1.0, y = 0.0, x' = \sqrt{3}/2, y' = 1/2$ and $\theta = \pi/6$. In specifying the values in this way, I mean to show the exact results that would be given by a computer with infinite precision rather than to imply that real numbers besides such as shown for x and y can be used in Aldat.

r1 <−[x', y'] **where** $x = 1.0$ **and** $y = 0.0$
and $\theta = \pi/6$ **in** *rotate*;

giving

r1(x'	y')
$\sqrt{3}/2$	1/2

r2 <−[x, y] **where** $x' = \sqrt{3}/2$ **and** $y' = 1/2$
and $\theta = \pi/6$ **in** *rotate*;

giving

r2(x	y)
1.0	0.0

r3 <−[x, y'] **where** $x' = \sqrt{3}/2$ **and** $y = 0.0$
and $\theta = \pi/6$ **in** *rotate*;

giving

r3(x	y')
1.0	1/2

r4 <−[x', y] **where** $x = 1.0$ **and** $y' = 1/2$
and $\theta = \pi/6$ **in** *rotate*;

giving

r4(x'	y)
$\sqrt{3}/2$	0.0

r5 <−[y', y] **where** $x = 1.0$ **and** $x' = \sqrt{3}/2$
and $\theta = \pi/6$ **in** *rotate*;

giving

r5(y'	y)
1/2	0.0

r6 <−[x, x'] **where** $y' = 1/2$ **and** $y = 0.0$
and $\theta = \pi/6$ **in** *rotate*;

giving

r6(x	x')
1.0	$\sqrt{3}/2$

r7 <−[θ] **where** $x = 1.0$ **and** $y = 0.0$
and $x' = \sqrt{3}/2$ **and** $y' = 1/2$ **in** *rotate*;

giving

r7(θ)
$\pi/6$

5. In the recursive domain algebra,

```
let Nom be Name ujoin
    [red ujoin of
        [red ujoin of Nom] in
        CHILDREN] in
    FAMILY;
```

Name will automatically be converted to a relation, since it is a scalar attribute:

```
let Nom be relation (Name) ujoin
    [red ujoin of
        [red ujoin of Nom] in
        CHILDREN] in
    FAMILY;
```

To show how this works, Table A abbreviates the data of Table 2 and adds another family. The notes explain each of virtual attributes as they appear from left to right.

B. Challenges to Query Languages

The significant advantage of Aldat over other database languages is that Aldat intends to *integrate* in one language all possible applications requiring secondary storage (as well as further applications, not necessarily on SS, which might benefit from a high level of language abstraction). The motivating examples of Section 2 and Appendix A indicate this, and I here list ten specific challenges to other secondary storage formalisms.

PERSON								**red**..
(*Name*	*FAMILY*	..)			**red**..	**relation**	*Nom*	(*Name*)^5
	(*Conj*	*CHILDREN* ..)		**red**..	(*Name*)^2	(*Name*)^3	(*Name*)^4	
		(.. *Name* ..)	*Nom*	(*Name*)^2				
			(*Name*)^1					
Ted	Alice	Mary	Mary	Mary	Mary	Ted	Ted	Ted
				James	James		Mary	Mary
		James	James	Mary	Pete		James	James
				James			Pete	Pete
	Sal	Pete	Pete	Pete	Mary			Ann
					James			Stu
					Pete			Mac
Ann	Tim	Stu	Stu	Stu	Stu	Ann	Ann	
				Mac	Mac		Stu	
		Mac	Mac	Stu			Mac	
				Mac				

Table A: recursive domain algebra.

Notes

1. At the leaf level, *Nom* is **relation** (*Name*) only: the second operand of the **ujoin** is empty.
2. Two anonymous reductions raise the level, using **red ujoin** to take unions.
3. **relation** (*Name*) two levels up.
4. *Nom* now is union of the results from notes 2 and 3.
5. Finally, a top-level
   [**red ujoin of** *Nom*] **in** *PERSON*
   gives the union of the results for `Ted` and `Ann`.

I do not believe that any one language apart from Aldat can do all of them, and most of them separately are beyond the capability of standard languages such as SQL and XML unless they have been augmented with special operators.

1. For a numeric attribute, N, of a relation (i.e., a column of a table), compute the standard deviation of its values (Section 4).

2. Represent two matrices as relations and compute their matrix product (Appendix A).

3. Represent as (a) relation(s) a square matrix (probably large and sparse) augmented by a column so it describes a system of linear equations, and solve those equations by Gaussian elimination [56].

4. Represent relationally a set of rules in the form
   if antecedent$_1$ **and .. and** antecedent$_n$
   then conclusion
   and a set of known facts (antecedents), and write an inference engine to derive all possible conclusions from the known facts (Appendix A).

5. Represent a map relationally and, given two regions, determine if they are adjacent [58, 62].

6. Compute all possible roll-ups for a datacube represented as a relation [55].

7. Represent relationally a set of shopping baskets (sets of items purchased at a supermarket) and compute association rules that say which items tend to be purchased together, with given significance and likelihood [55].

8. Represent the composition of a manufactured product (assembly, subassembly, quantity) as a relation and compute the "transitive closure", i.e., the total quantities of each subassembly needed for the product and for any intermediate assembly (Appendix A).

9. Compute a relation, or the XML equivalent, which describes all the paths in a nested relation, or the XML equivalent [57].

10. In a recursively nested relation, or the XML equivalent,
    PERSON(*Name, FAMILY*(*Spouse,*
    CHILDREN(*Name, FAMILY*(..))))
    find the Name of the PERSON who has a youngest descendant named Mary but no intermediate descendants named Mary. Alternatively, find the Name of the PERSON who has an unbroken chain of descendants named Mary [59].

C. Chronological Glossary

This appendix summarizes Aldat facilities, giving cross-references within the paper, citations and chronology. (Not every keyword originally had the name used in this paper.)

Keywords	Year	Citation	Section
=	~1976	[47]	5.1
[ ]	~1976	[47]	5.2
djoin	~1976	[47]	5.1
icomp	~1976	[47]	5.1
ijoin	~1976	[47]	5.1
in	~1976	[47]	5.2
ljoin	~1976	[47]	5.1
natcomp	~1976	[47]	5.1
natjoin	~1976	[47]	5.1
propsub	~1976	[47]	5.1
propsup	~1976	[47]	5.1
rjoin	~1976	[47]	5.1
sep	~1976	[47]	5.1
sjoin	~1976	[47]	5.1
sub	~1976	[47]	5.1
sup	~1976	[47]	5.1
ujoin	~1976	[47]	5.1
where	~1976	[47]	5.2
equiv	~1977	[51]	4, 6
fun	~1977	[51]	4
let .. be	~1977	[51]	4
par	~1977	[51]	4
red	~1977	[51]	4, 6
#	1978	[48]	5.2
•	1978	[48]	5.2
quant	1978	[48]	5.2
add	~1983	[50]	5.5, 7
change	~1983	[50]	5.5, 7
delete	~1983	[50]	5.5, 7
on	~1983	[50]	5.5, 7
update	~1983	[50]	5.5, 7
using	~1983	[50]	5.5
is	~1984	[14]	5.4
when	1991	[21]	5.2
pick	1991	[21]	5.2
comp	1993	[53]	7
in	1993	[53]	7
out	1993	[53]	7
events	1996	[38]	7
pre	1996	[38]	7
post	1996	[38]	7
nesting	1998	[87]	6
relation()	1998	[87]	6
aldatp	~2000	[]	8.2
eval	2001	[55]	6
quote	2001	[55]	6
transpose	2001	[55]	6
state	2002	[90]	7
recursive nesting	2004	[86]	6
path expression	2005	[57]	6
display2D	2005	[91]	5.2
grep	2005	[57]	5.2

REFERENCES

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : from Relations to Semistructured Data and XML.* Morgan Kaufmann, San Francisco, 2000.
2. M. **Andreev**. Operations on text in a database programming language. Master's thesis, McGill University, School of Computer Science, Nov. 1999.
3. M. P. Atkinson and R. Morrison. *Persistent First Class Procedures are Enough*, volume 181 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1984.
4. P. Baker. Design and implementation of database computations in java. Master's thesis, McGill University, School of Computer Science, July 1998.
5. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–17, September 1975.
6. Robert S. Boyer and J. Strother Moore. A fast string search algorithm. *Communications of the ACM*, 20(10):762–72, Oct. 1977.
7. D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle connection or UNIXes of the world unite! *Software — Practice & Experience*, 12(12):1147–62, December 1982.
8. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
9. D. D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.
10. Andy **Chang**. Implementation of sigma-joins in a nested relational language. M.Sc. project report, July 2002.
11. YuLing **Chen**. A G.I.S. editor for a database programming language. Master's thesis, McGill University, School of Computer Science, March 2001.
12. Ann Chong. Implementation of the mu-joins in relix. M.Sc. project report, Sept. 1986.
13. A. Clouâtre. *Implementation and Applications of Recursively Defined Relations.* PhD thesis, McGill University, School of Computer

Science, 1987.

14. A. Clouâtre, N. Laliberté, and T. H. Merrett. A general implementation of relational recursion with speedup techniques for programmers. *Information Processing Letters*, 32:257–61, 22 Sept. 1989.
15. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, June 1970.
16. E. F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, pages 34–64. Prentice-Hall, Engelwood Cliffs, N. J., 1972. Courant Institute of Mathematical Sciences, New York University, 1971/5/24–25.
17. E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, Engelwood Cliffs, N. J., 1972. Courant Institute of Mathematical Sciences, New York University, 1971/5/24–25.
18. 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.
19. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G+: Recursive queries without recursion. In L. Kershberg, editor, *Proc. of the Second Internat. Conf. on Expert Database Systems*, pages 355–68, Tysons Corner, Va., April 25–7 1988.
20. R. de la Briandais. File searching using variable-length keys. In *Proc. Western Joint Computer Conf.*, pages 295–8, San Francisco, March 1959.
21. S. Douik. Implementation of delay and nondeterminism in a database programming language. Master's thesis, McGill University, School of Computer Science, October 1991.
22. Bernhard Düchting. A relational picture editor. Master's thesis, McGill University, School of Computer Science, Aug. 1983.
23. Brenda Fayerman. A text editor based on relations. Master's thesis, McGill University, School of Computer Science, Aug. 1984.
24. E. H. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–9, Sept. 1960.
25. I. **Garton**. Concurrency in B-trees and tries. Master's thesis, McGill University, School of Computer Science, June 2000.
26. G. H. Gonnet. private communication. ∼1989.
27. 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 sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997.
28. R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In Ian Munro, editor, *SODA'04 Proceedings of the fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 636–45, New Orleans, Jan. 2004.
29. Yu **Gu**. Basic operators for semistructured data in a relational programming language. Master's thesis, McGill University, School of Computer Science, Dec. 2005.
30. B. Gunnlaugsson. Concurrency and sharing in prolog and in a picture editor for aldat. Master's thesis, McGill University, School of Computer Science, May 1987.
31. Fan **Guo**. Implementing attribute metadata operators to support semistructured data. Master's thesis, McGill University, School of Computer Science, Jan. 2005.
32. B. Hao. Implementation of the nested relational algebra in java. Master's thesis, McGill University, School of Computer Science, March 1998.
33. H. He. Implementation of nested relations in a database programming language. Master's thesis, McGill University, School of Computer Science, July 1997.
34. X. X. Hu. Implementation of a constraint-based graphics editor for relix. Master's thesis, McGill University, School of Computer Science, Nov. 1991.
35. 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.

36. Sungsoo **Kang**. Implementation of functional mapping in a nested domain algebra. M.Sc. project report, Sept. 2001.
37. Alan Kay. personal communication. 1996.
38. A. El Kays. Implementation of event handlers in a database programming language. Master's thesis, McGill University, School of Computer Science, March 1996.
39. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume I. Addison-Wesley Publishing Co., Reading, Mass., 1997. 3rd edition.
40. D. E. Knuth, J. H. Jr Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–50, 1977.
41. M. Komioti. Implementation of process synchronization and other operators in a database programming language. Master's thesis, McGill University, School of Computer Science, March 1992.
42. N. Laliberté. Design and implementation of a primary memory version of aldat, including recursive relations. Master's thesis, McGill University, School of Computer Science, 1986.
43. R. Lui. Implementation of procedures in a database programming language. Master's thesis, McGill University, School of Computer Science, Aug. 1996.
44. 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.
45. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In David Johnson, editor, *SODA'90 Proceedings of the first Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–27, San Francisco, Jan. 1990. Society for Industrial and Applied Mathematics.
46. A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1(1):54–67, April 1997. (Also *Proc. PDIS*, 1996).
47. T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, Feb. 1977.
48. T. H. Merrett. The extended relational algebra, a basis for query languages. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 99–128. Academic Press, 1978.
49. T. H. Merrett. Multidimensional paging for efficient data base querying. In *International Conference on Data Base Management Systems*, pages 277–90, Milano, June 1978.
50. T. H. Merrett. *Relational Information Systems.* Reston Publishing Co., Reston, VA., 1984.
51. 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.
52. T. H. Merrett. Relixpert — an expert system shell written in a database programming language. *Data and Knowledge Engineering*, 6:151–8, 1991. Fuller version available as technical report of same name: McGill University, School of Computer Science, TR–SOCS–89.4, July, 1988.
53. T. H. Merrett. Computations: Constraint programming with the relational algebra. In A. Makinouchi, editor, *Proceedings of the International Symposium on Next Generation Database Systems and Their Applications*, pages 12–17, Fukuoka, Japan, September 28–30 1993. *Invited Paper.*
54. T. H. Merrett. Relational information systems. (revisions of [50]):
    Data structures for secondary storage http://www.cs.mcgill.ca/∼cs420
    Database programming:
    http://www.cs.mcgill.ca/∼cs612, 1999.
55. T. H. Merrett. Attribute metadata for relational OLAP and data mining. In P. Manghi and G. Ghelli, editors, *Proceedings, 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*, pages 65–76, Monteporzio Catone — Roma, Italy, Sept. 8–10 2001. www.cs.mcgill.ca/˜tim/dbpl/attrMeta.ps, 235K.

56. T. H. Merrett. CS++: Reinventing computer science (for secondary storage). Technical Report TR-Ald05-01, School of Computer Science, McGill University, Montreal, Feb. 2005. URL: www.cs.mcgill.ca/∼tim/CS++.pdf (750K).
57. T. H. Merrett. A nested relation implementation for semistructured data. Technical Report TR-Ald05-03, School of Computer Science, McGill University, Montreal, April 2005. URL: www.cs.mcgill.ca/∼tim/semistruc/recnest.ps.gz.
58. T. H. Merrett. Quad-edge data structures in two and three dimensions. Technical Report TR-Ald05-02, School of Computer Science, McGill University, Montreal, April 2005. URL: www.cs.mcgill.ca/∼cs617/handouts/quadedge050329.pdf.
59. T. H. Merrett. Semistructure from relations. Technical Report TR-Ald05-04, School of Computer Science, McGill University, Montreal, May 2005. URL: www.cs.mcgill.ca/∼tim/semistruc/rel2semi.pdf.
60. T. H. Merrett. Notes on semistructured data. www.cs.mcgill.ca/∼tim/semistruc/, 2006.
61. T. H. Merrett. Working notes on tries. www.cs.mcgill.ca/∼tim/tries/, 2006.
62. T. H. Merrett, Y. Bédard, D. J. Coleman, J. Han, B. Moulin, B. Nickerson, and C. V. Tao. A tutorial on database technology for geospatial applications. www.cs.mcgill.ca/∼tim/geodem/tutorial.ps, 2002.
63. T. H. Merrett and E. J. **Otoo**. Dynamic multipaging: a storage structure for large shared data banks. In P. Scheuermann, editor, *Improving Database Usability and Responsiveness*, pages 237–56, New York, 1982. Academic Press.
64. T. H. Merrett and H. **Shang**. Trie methods for representing text. Technical Report TR–SOCS–93.5, McGill University, School of Computer Science, June 1993.
65. T. H. Merrett and H. **Shang**. Trie methods for representing text. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms*, pages 130–145, Chicago, Illinois, October 13–15 1993. Springer-Verlag GmbH. LNCS 730.
66. T. H. Merrett and H. **Shang**. Zoom tries: A file structure to support spatial zooming. In T. C. Waugh and R. G. Healey, editors, *Advances in GIS Research Proceedings, Volume 2*, pages 792–804, Edinburgh, Scotland, September 5–9 1994. Taylor & Francis.
67. E. Mnushkin. Inheritance in a relational object-oriented database system. Master's thesis, McGill University, School of Computer Science, March 1992.
68. D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–34, 1968.
69. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric, multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.
70. J. A. **Orenstein**. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–7, June 1982.
71. J. A. **Orenstein**. *Algorithms for Implementing Relational Databases*. PhD thesis, McGill University, School of Computer Science, Jan. 1983.
72. J.A. **Orenstein**. Blocking mechanism used by multidimensional tries. Unpublished Letter, February 1983.
73. E. J. **Otoo**. *Low Level Structures in the Implementation of the Relational Algebra*. PhD thesis, McGill University, School of Computer Science, Aug. 1983.
74. G. Peano. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.*, 36:157–60, 1890.
75. Andrey **Rozenberg**. Implementing attribute metadata with application to data mining. M.Sc. project report, Jan. 2002.
76. J. W. Schmidt. Some high-level constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–61, September 1977.
77. Heping **Shang**. *Trie Methods for Text and Spatial Data on Secondary Storage*. PhD thesis, School of Computer Science, McGill University, January 1995. URL: www.cs.mcgill.ca/∼tim/cv/students.html.
78. W. **Sun**. Updates and events in a nested rela-

tional programming language. Master's thesis, McGill University, School of Computer Science, March 2000.

79. N. Sutyanyong. Procedural abstraction in a relational database programming language. Master's thesis, McGill University, School of Computer Science, June 1994.
80. S. J. P. Todd. The Peterlee relational test vehicle—a system overview. *I.B.M. Systems Journal*, 15(4):285–308, 1976.
81. M. Tsakalis. Implementing QT-selectors and updates for a primary memory version of aldat. Master's thesis, McGill University, School of Computer Science, March 1987.
82. A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Informat. Theory*, IT-13:260–9, April 1967.
83. Zongyan **Wang**. Implementation of distributed data processing in a database programming language. Master's thesis, School of Computer Science, McGill University, Nov. 2002. URL: www.cs.mcgill.ca/∼tim/cv/students.html.
84. Catharine M. Wyss and Edward L. Robertson. Relational languages for metadata integration. *ACM Trans. on Database Systems*, 30(2):624–60, 2005.
85. Jiantao **Xie**. Text operators in a relational programming language. Master's thesis, McGill University, School of Computer Science, Jan. 2005.
86. Zhan **Yu**. Implementation of recursively nested relations. M.Sc. project report, Jan. 2004.
87. Z. Yuan. Java implementation of the domain algebra for nested relations. Master's thesis, McGill University, School of Computer Science, Aug. 1998.
88. Hongyu **Zhao**. Implementation of quantified selection in a nested relational language. M.Sc. project report, April 2002.
89. X. Y. **Zhao**. *Trie Methods for Structured Data on Secondary Storage.* PhD thesis, McGill University, School of Computer Science, June 2000. URL: www.cs.mcgill.ca/∼tim/cv/students.html.
90. Yi **Zheng**. Abstract data types and extended domain operations in a nested relational algebra. Master's thesis, McGill University, School of Computer Science, Aug. 2002. URL: www.cs.mcgill.ca/∼tim/cv/students.html.
91. Lili **Zhu**. A generalized two-dimensional display editor for relations. Master's thesis, McGill University, School of Computer Science, Dec. 2005.