AN ALGEBRA OF RELATIONS FOR MACHINE COMPUTATION

the paper that should have been the game changer, with annotations by Hugh Darwen, 2014

Patrick Hall, Peter Hitchcock, Stephen Todd
IBM (UK) Scientific Centre
Neville Road
Peterlee
Co. Durham
England

Abstract:

This paper extends the relational algebra of data bases, presented by Codd [4] and others, in four areas. The first is the use of selector names to remove order dependencies from the columns of a relation. This use of selector names enables us to define a more general class of operations, which include the normal relational operations of union, equijoin etc., as special cases. Thirdly we introduce relations represented algorithmically as well as by a stored set of tuples. Such computed relations cannot always be effectively realised as a finite set of tuples. Finally we consider relational expressions as algorithmic representations of relations and characterize their effectiveness.

HD: I suggest that this paper *would* have been a game changer—showing the right way forward with the relational model—if only it and its aftermath in software development at the IBM UK Scientific Centre had received wider attention than apparently it did.

The original text was in the usual two-column format and in something like Courier throughout. In other respects I have tried to retain the original formatting.

I believe I read this paper in 1978, when my colleague Brian Rastrick and I visited Peterlee to learn about ISBL, the language, and PRTV, its implementation, but I didn't know much about logic and set theory at that time and much of the formal text would have been over my head. Instead, I was heavily influenced by the language, ISBL, whose relational operators are based on the abstract algebra defined herein, in the same way that the relational operators of **Tutorial D** [16] are defined in terms of the abstract algebra, **A**, defined by Chris Date and myself [16] and first published in 1998.

The first implementation of ISBL, named IS/1, was developed at the Scientific Centre as early as 1972. IS/1 was later renamed Peterlee Relational Test Vehicle (PRTV) at the behest of executive management at IBM.

I have shaded in yellow some points I find particularly noteworthy.

In my notes, labelled **HD1** to **HD23**, the abbreviations HHT, *TTM* and BS12 refer to the subject paper by Hall, Hitchcock and Todd, *The Third Manifesto* [14], and Business System 12 [15], respectively.

Acknowledgments: Stephen Todd reviewed the first draft of my annotations and in so doing gave me some additional material as well as correcting a couple of errors. Chris Date did the same.

1. Motivation.

We may think of a relation as a table with rows and columns. Row ordering is unimportant but in the relational algebras developed so far a knowledge of column ordering has been necessary to specify certain operations. When tables have many columns, this can be tedious. Codd [4] discusses the use of what he calls relationships, where the components of a tuple or relation are identified by role name and not position. The formal details of relationships were not followed up by Codd, as he was only interested in them for user convenience. He does not seem to have considered the role names of the result of a relational expression. This is not a straightforward problem. For example, given a relation with role names A and B identifying the columns, if we use the project operator to define a new two column relation which has both columns identical to the original A column, the role names cannot be simply inherited. Problems concerning the use of role names have not been fully realised in the literature, let alone adequately solved. Our solution involves generalizing the relational operations so that the use of role names controls the semantics of the operations. The particular method we present below is one approach: we have concurrently also developed an approach based on characteristic functions and lambda notation.

HD1 The "role names" mentioned here are what soon became known as attribute names. The authors identify the very problem that BS12 planners (myself and Brian Rastrick) were confronted with in 1978. We had been assuming all along, from Chris Date's teaching, that attribute names would be used to identify "columns" but couldn't figure out how the operators of Codd's algebra would work and we found serious problems with the use of "dot qualifiers" in his calculus notation. Codd did later consider the "column naming" problem and attempted to address it; but his treatment was rather ad hoc and in fact seriously deficient (see [19], pages 148-152).

The example given, suggesting that the project operator can be used to duplicate a column, will no doubt give cause for surprise. The authors are assuming the definition of "generalized project" (Defn 3.5 in Section 3), which incorporates attribute renaming and allows the same attribute to be "renamed" more than once, so to speak. But in that case, all except one of those "renamings" is really an extension.

HD2 The remaining text in this section begins to address the other big concern we had in 1978: given a relation with attributes qty and price, for example, how to obtain a relation that gives the total cost (qty*price) for each item. Codd's algebra did not address this issue.

The second area we have attacked has been that of including functional operations on relations. We only consider here first order functions (those which take a row of a relation and produce a row of a result relation) as opposed to second order ones which act across tuples, see Hitchcock [9]. Such operations have been easy to specify in the relational calculus, due to the ability to reason with free variables. Consider:

$$C = \{< a,b,c > : b = \sin(a) \ \& \ <a, c> \text{in } D\}$$

We would define such an operation on D by defining a relation SINE using an algorithm, and combine SINE and D using one of our generalised operations. While considering first order operations on relations, we have also considered the storing of

sets as procedures which would successively generate the elements of the set—for example, the set INT of non-negative integers.

Algorithmic relations cannot be used as freely as stored relations. For example:

(i) a typical relation SINE may only be used to compute the sine given the angle, but not the inverse.

(ii) the relations SINE and INT are potentially infinite and the evaluation of relational expressions involving them may not terminate.

2. Relations with selectors.

HD3 HHT now uses the term selector for the attribute names that Section 1. Motivation referred to as role names. (*TTM* uses "selector" for something completely different.)

We develop a precise notion of relation. In normal usage, this is done via set, tuple, and cartesian product. We take a parallel approach via set, tuple with selectors, and cartesian product with selectors, and define a relation with selectors which is equivalent to Codd's 'relationship' . Because we deal only with tuples with selectors we will drop the 'with selectors' after the initial exposition.

We assume a universal underlying set U. The elements of U are called objects. In a working system we would further divide the objects into 'domains'—this gives additional complications with compatibility of operators, etc. which we will not consider here. However we will discuss multiple domains with respect to their effectiveness in later sections.

HD4 These "objects" are what *TTM* refers to as values. HHT does not mention whether relations are also objects—i.e., whether a relation can be an attribute value—but whether they are or not has no effect on the basic algebra. I assume that the domains mentioned here are what we now call types. The kinds of operations in which relations as attribute values become relevant—grouping and aggregation, for example—are not mentioned in HHT, though aggregation via grouping was later supported in PRTV.

Defn. 2.1: a tuple, t, with selectors, S, is a function t:S→U. S is the selector set for t, and an element of S is a selector name for t. The elements of the range of t are called the objects of t. t(s) is called the value of s for the tuple t. □

HD5 *TTM* defines a tuple as a set of ordered triples, $<a, v, t>$, where *a* is an attribute name, *v* a value, *t* the type of *v*, and no two triples within the same tuple have the same *a*. As *t* is implied by *v*, *TTM*'s definition is entirely consistent with HHT.

Defn. 2.2: the cartesian product U(S) with selectors S is the set of all tuples with selectors S. □

HD6 *TTM* calls this set a tuple type.

Defn. 2.3: a relation with selectors S is a set of tuples with selectors S. Equivalently a relation with selectors S is a subset of the cartesian product with selectors S. □

HD7 *TTM* uses the term *body* for the set of tuples and defines a relation as a heading paired with a body, the heading being a function of S to type names

which merely incorporates the "domains" that the authors have identified as being needed in a "working system".

Defn. 2.4: the degree of a relation (tuple, cartesian product) with selectors S is the number of elements of S. □

Defn. 2.5: the cardinality of a relation is the number of tuples in the relation. □

HD8 *TTM*'s definitions of degree and cardinality are equivalent to HHT's.

Example 2.1:
Consider the relation R, represented by the table

Cust no.	part no.	qty.
123	49	3
241	74	7
333	33	3
123	50	4

The selector set for R is cust no, part no, qty. The degree is 3, the cardinality 4. R contains a tuple, t, which we can represent by:

Cust no.	part no.	qty.
123	49	3

This tuple might mean that customer number 123 has ordered 3 of part number 49. We can identify the quantity object for this tuple using the selector name qty, because t(qty) = 3.

HD9 In **Tutorial D** t(qty) becomes qty FROM t.

It is important to note that not only is the row ordering of the table representation of R unimportant, the column ordering also does not matter. It is not meaningful to talk of the third column of a relation with selectors, rather we talk of the qty column. Also the column headings must be carried into a representation of the tuple.

HD10 That last sentence is consistent with *TTM*. The point about column ordering was crucial for BS12. Prerelational scripting languages for queries and reports used field names for record components and scripts did not depend at all on the order in which the fields appeared. To introduce such a dependency would have been a backward step. (For "column headings", read "selectors", i.e., attribute names again.)

3. The Generalised Operators.

We begin with a definition of all the operators, and then explain them in terms of the usual relational operators. Finally we discuss the algebra induced by these operations.

For all the definitions, we assume that the relations R1, R2, Q, have selector sets S1, S2, P, respectively.

Notation: $t|S$ means function t restricted to the subset S of its domain; $t \circ q$ is functional composition, and is the function whose application to object x is equivalent to t(q(x)); S1.S2 means the usual set intersection of sets S1 and S2.

Defn. 3.1. Generalised Intersection.

R1*R2
= {t: t in U(S1∪S2) & $t|Sl$ in R1 & $t|S2$ in R2}.

□

HD11 Generalised intersection is the ◂AND▸ operator of the abstract algebra **A** defined by Chris Date and myself and also **Tutorial D**'s JOIN (Codd's "natural join").

Defn, 3.2. Generalised Union.
R1 + R2
= {t: t in U(S1∪S2) & (t | Sl in R1 ∨ t | S2 in R2)}.
□

HD12 Generalised union is the ◂OR▸ operator of **A**. Note that if some attribute of either operand is of an infinite type and is not a common attribute, then *r1* ◂OR▸ *r2* is infinite. This issue is discussed in Section 4 of HHT.

Defn. 3.3. Project.
Let P be contained in S1,
R1 | P ={ t': t'=t | P & t in Rl }
□

HD13 In **A**, projection is defined in terms of an attribute of the relation operand that is to be *ex*cluded: *r* ◂REMOVE▸ *A* denotes the projection of *r* on all attributes except *A*. BS12 and **Tutorial D** both allow projection to be defined either way: attributes to be retained or those to be dropped—but this is in any case only a matter of convenience.

Defn. 3.4. Generalised Difference.
R1 - R2
={t: t in R1 & t | (S1.S2) not in (R2 | (S1.S2))}
□

HD14 **A** supports negation by defining ◂NOT▸ *r* to be the relation complement of *r:* in HHT terms, the tuples of U(S) that are not contained in *r*. This is not suitable for computational purposes because when an underlying domain is infinite, it yields an infinite relation. HHT's R1-R2 can be defined as R1 ◂AND▸ (◂NOT▸ R2), which becomes R1 NOT MATCHING R2 in **Tutorial D**.

David Maier [17] refers to this operator as *antijoin*. The term *semidifference* has also been advanced for it, as being the opposite of *semijoin* (MATCHING in **Tutorial D**), a shorthand for which ISBL did not have a counterpart.

Defn. 3.5. Generalised Project.
Let q be a function from selector set P to selector set S1,
Rl%q
= {t': t' in U(P) & (there exists t in Rl: t'=t∘q) }.
□

HD15 An element of q might be written as $a \rightarrow b$, where *a* is a selector of R1 and *b*, if not equal to *a*, is a selector that is not a selector of R1. Thus we can see here, in addition to projection, the beginnings of the operators that later came along for attribute renaming and extension (RENAME and EXTEND in **Tutorial D**—note that RENAME can be defined in terms of extension followed by projection).

BS12's projection operator was equivalent to generalised project except that q was required to be injective, meaning it could not be used to "duplicate columns". Here "selector set P" appears to be an arbitrary set of selectors, not as in Defn. 3.3, where it is "contained in S1".

The generalised intersection is probably the most useful operator. When R1 and R2 are of the same type (ie. S1=S2), the generalised intersection is set intersection. At the other extreme, if S1 and S2 do not overlap, we have defined a cartesian product or 'quadratic join' of R1 and R2. In between, we have an equijoin [4] of the two relations, the joining components being the selectors common to both R1 and R2 (i.e. S1 . S2).

HD16 Actually, it was Codd's "natural join", not equijoin, that used the common selectors. The equijoin of *r1* and *r2* on *A=B*, where *A* is an attribute of *r1* and *B* an attribute of *r2,* retains both attributes *A* and *B*—and so gets into trouble when *A* and *B* have the same name!

Generalised intersection can also cover the notion of select. Suppose we have a selection criterion or 'filter' defined on certain selectors. We can define a relation F (possibly infinite) with selector set containing just those selectors, and whose tuples are just those tuples which satisfy the filter. Then the selection on a relation R using the filter is the same as the generalised intersection between the relation R and the relation F derived from the filter.

HD17 This observation is emulated in the description of **A**, which does not include a selection or restriction operator. It hints at the idea, discussed later in the paper, of allowing the selection condition to use whatever boolean operators are available, in any combination. A similar observation applies to the next paragraph, hinting at the possibility of the operator we now call extension.

Another use of intersection is functional application. A relation, viewed as a function mapping a subset of its components to the remaining components, can be applied to a set of arguments to yield a set of results.

Generalised union degenerates to normal union when R1 and R2 have the same selectors. When the selectors are different, generalised union seems to have useful properties related to undefined values and the merging of heterogeneous files. We do not fully understand the implications of these yet.

HD18 A commendably cautious treatment, considering what happened later when outer joins were added SQL!

Generalised project has several uses. It can be used to duplicate components, but more usefully to rename components that are to be used in a join. It is informally called rename. For example, suppose we have relations, one of which has a component date-due, and the other of which has a component date-returned. If we wished to join on these components (to find which books were returned on the last possible day), we would need to rename at least one component so that the names were identical. Generalised intersection would then perform the join.

HD19 Regarding its use "to duplicate components", see HD14. In **Tutorial D** we define separate operators for projection and attribute renaming. Duplicating a component is done with EXTEND. I understand that in the later versions of PRTV projection, renaming and extension were all combined in a single operator, as in SQL's SELECT clause. This idea was considered for BS12 but

was not adopted because we wanted multiple attribute renamings to be simultaneous, in parallel—to allow names to be swapped, for example—whereas we wanted multiple extensions to be done consecutively, so that, for example, { y := x +1, z := y * 2 } and { y := x +1, z := (x + 1) * 2 } would be equivalent.

Project is a special case of the generalised project, where q is the inclusion mapping of p into S1. We have included it for notational convenience.

Finally we come to generalised difference. Once again, if R1 and R2 have the same selectors, then this degenerates to set difference. As an example of its more esoteric use, let R1 and R2 represent a hierarchy with R1 containing the tuples of the parent segments, R2 the tuples of the child segments, including the fully concatenated key of the parent. The selectors corresponding to the fully concatenated key are the only overlap of S1 and S2. Then R1-R2 gives the childless parent records, and R2-R1 gives the orphaned child records.

Using these operations we can combine relations within expressions. Each expression defines a relation. The selectors of this relation can be readily obtained from the selectors of the operand relations as given below:

selectors(Rl*R2) = selectors(R1) ∪ selectors(R2)
selectors(Rl+R2) = selectors(R1) ∪ selectors(R2)
selectors(Rl | P) = P
selectors (R1-R2) = selectors(R1)
selectors (Rl%q) = domain(q)

HD20 The choice of "domain(q)" in that last definition is a little unfortunate, considering the previous references to Codd's use of the term domain. Here it appears to mean the domain of the function q, which is indeed a set of selectors.

Relations with selectors, together with the binary operations (generalised) union, intersection, and difference, form a Boolean Algebra. To demonstrate this structure, we require a universal element, a null element, partial ordering relationships, and must then verify the various idempotency and other algebraic laws. For the universal element, we suppose that all selectors are drawn from a set Σ of selectors, when the universal element is U(Σ). The null element is the empty relation of no selectors (U(ϕ) is the cartesian product of no selectors, which is either empty or full, these two conditions being effectively the truth values).

HD21 That last sentence delightfully hints at what many years later became TABLE_DEE and TABLE_DUM. I question "being effectively the truth values". A relation is a relation and a truth value is a truth value. Under the interpretation of a relation as the extension of a predicate, a relation of degree zero is the extension of a predicate with no parameters, which therefore has just one instantiation. The empty relation denotes falsehood for that single instantiation whereas the "full" one (containing a single tuple) denotes its truth.

The ordering relationship is relation inclusion, where a relation R1 contains another relation R2 if the selectors of R1 include the selectors of R2, and the difference R2-R1 is empty. The various algebraic laws then follow, with universal complement obtained from generalised difference.

4. The Representation of Relations and their Effectiveness.

HD22 Sections 4, 5, and 6 are rather heavy going, a lengthy investigation leading to the important conclusions, in Section 7, that I alluded to in **HD2**. We might even conjecture that the undue amount of text on this particular topic got in the way of the most important points of this paper, thereby reducing its impact back in 1975.

I don't give any annotations on these three sections. Rather, in **HD23** in Section 7, Conclusions, I try to show what the important hints given in that section were really driving towards.

When we come to embody relations in computing machinery, we find two extremes. At the one end of the spectrum we have relations which are stored explicitly as sets of tuples, and at the other end we have relations which are pure predicates, ie. we can only recognise if a given tuple is in the relation or not. (We do not consider those relations whose characteristic functions are not decidable). In the Preceding sections defining relations and the operations on them we deliberately do not distinguish between such relations. However they have very different properties when it comes to generating their contents, as the following examples make clear. What we wish to do in the following sections is to explore the different behaviour of such relations, and expressions involving them, when we try to generate them.

Examples :

Ex. 4.1: If we are given a representation of a relation which is a table, as in Ex. 2.1, then we can generate all of the tuples in the table without additional input. These are the relations as they are usually conceived in relational data bases.

Ex. 4.2: We have a binary relation, SINE, between an angle and its sine, which is represented by a procedure to calculate the sine of a given angle. We cannot generate the extent of the relation itself, although we can generate the result of a relational expression which applies it to a finite set of angles. We cannot, however, apply the representation of sine so that we obtain arcsines.

Ex.4.3: If we have a relation which is a pure predicate, such as a test whether a given point is in a particular polygon, then we require that it is used in relational expressions as a pure predicate. We call such representations of relations recognisers.

Ex. 4.4: Returning to ex. 4.2, we could consider the same representation augmented by a second procedure which computes the arcsine, thus allowing the representation to be used in both directions. Note that this still does not give a generation capability.

We will use the term 'effectiveness' to talk about the generation properties of the representations of relations and relational expressions.

We will characterise the effectiveness of a representation in terms of a set of input sets. We will call a subset of the selectors of a relation an 'input set' for the representation if, given values for these selectors we can then obtain a complete set of values. We gather together all such input sets into a set of input sets which is the effectiveness of the representation. Note that if any set of selectors P is an input set, then any set Q containing P is also an input set.

Returning to our examples above, we now characterise the effectiveness of their representations using sets of input sets. We write the effectiveness of a representation of relation R as E(R).

Ex. 4.5: for Ex. 4.1 we have
E(R) = { ϕ , {cust no}, {part no}, {qty}, { cust no, part no }, {cust no, qty } , {part no, qty }, {cust no, part no, qty}}.

This reads 'given no inputs, or any set of inputs, we can obtain all the other components.'

Ex. 4.6: for Ex. 4.2 we have
E(SINE) ={x}, {x,sx}}.

This reads 'given the value of the angle x, we can obtain the value of the other component, namely the sine sx of the angle; or given both angle x and number SX, we can test whether sx=sin(x)'.

Ex. 4.7: for Ex. 4.3 we have
E(POINT-IN-POLYGON) = {{point, polygon}} .

This reads 'given a point and a polvgon, we can test whether the point is in the polygon'.

Ex. 4.8: for Ex. 4.4 we have
E(SINE)= {{X}, {SX, }{ X,SX}}.

This reads 'given the angle, we can compute the sine, or, given the sine, we can compute the angle, or, given both, we can test for membership' .

To use effectiveness in a particular case, we find the set P of selectors for the input values. If P is in the effectiveness, we know that the representation will be adequate for this case.

The view of relations as mappings from a subset of the components to another subset of the components is by no means new. It has even occurred within relational data bases, as the 'mapping' concept of SQUARE [1] and SEQUEL [2] , where it is used as a user-friendly language device; and it has appeared as a means of representing functional constraints within 'normalisation! [5,12].

5. The Effectiveness of Expressions.

When relations are combined by relational operators within an expression, the result is also a relation. If we have representations of the operands, then the relational expression together with these representations forms a representation of the result. Given the effectiveness of the representations of the operands, we can deduce the effectiveness of the expression.

We require, for each relational operation, to specify the effectiveness of the result as a function of the effectiveness of the operands. In the following we will write the selectors (R1)=S1, the selectors (R2)=S2, so that the two relations RI and R2 have selectors S1△S2, the symmetric difference between S1 and S2, in common.

E(R1*R2) ={P : (P.S1 in E(R1) and (P∪Sl).S2in E(R2))
or (P.S2 in E(R2) and (P∪S2).S1in E(R1))}.

E(R1+R2) = {P : P.Sl inE(Rl) and P.S2 in E(R2) and P contains S1△S2}.

E(R1 P) = { Q:Q contained in P, and Q in E(Rl)}.

E(R1-R2) = E (R2) if S1.S2 in E(R2) ϕ otherwise.

E(Rl%q) = {P:q(P) in E(Rl)}.

Note that the last three operations all involve quantification, and that in some cases it is possible that the operation is totally ineffective, with no input sets at all.

To see that these meet our intuition concerning the relational operations, we require some notion concerning the evaluation mechanism for relational expressions . The evaluation would be done 'tuple at a time', and given the values of some of the selectors, the mechanism would attempt to find values for the remaining selectors by requesting a complete tuple from one of the operand sub-expressions and then use the extra information so gained to fill in the remaining values from the other operand, ensuring that the constraints required by the particular operation are met before returning the completed tuple to the invoking procedure.

For intersection we could use the given values on either operand with equal effectiveness, and the values gained from one operand can be input into the other to obtain the complete tuple.

Ex 5.1: we have relation INT with selector x and effectiveness{ϕ, { x}}, and relation SINE as in Ex. 4.6. Then to evaluate INT*SINE (which has selectors x and sx) using no inputs, we must look to either INT or SINE for a complete tuple. INT can provide it, and this value x can then be supplied to SINE to obtain the sx value. The effectiveness of INT*SINE from the formula above is {ϕ , {x}, {sx}, {x, sx}}, which tells us we have a generator, and agrees with the description of how we would actually evaluate the expression.

For union we are not able to carry over values gained from one operand to help us evaluate the other operand, and any input values must be used independently on both operands.

Ex. 5.2: suppose that in addition to the SINE relation already introduced, we have a relation COSINE with selectors x and cx and effectiveness {{x }, {x, cx }}. Then SINE+COSINE with selectors x, sx, and cx, contains all tuples <x, sx, cx> where either sx=sin(x) or cx=cos(x). To use this relation, supplying a value for x is not enough. Given the value of x we can find one tuple from the relation by applying SINE and COSINE separately, but we cannot find all the tuples with this particular x. We only require that one of the two relations is satisfied, so for example applying SINE to x would fill in the sx value, to which we would then need to add all possible values for cx. We could do this last by using a representation of the underlying set; this possibility will be investigated in the next section; here we assume that the underlying set cannot be used, and so we are unable to realise the relation SINE+COSINE given only an x value.

However, if we supply a pair of values, say x and sx, then we may be able to fill in everything—if it happens that sx is not equal to sin(x), then applying COSINE to x will complete the tuple. But if sx=sin(x), we are back in the earlier situation, and cannot realise the relation.

The only circumstance under which we can effectively use the relation SINE+COSINE is if we are given all three values, and the relation is used as a pure recogniser. From the formulae, we find that the effectiveness of SINE+COSINE is {{x, sx, cx }}, which tells us that we only have a recogniser.

There are examples where union does provide more than a recogniser, such as the combination of SINE and COSINE where the selector cx of COSINE has been renamed sx. Then SINE+COSINE has selectors x and sx and has effectiveness {{x}, {x, sx}}.

When generalised union degenerates to a normal union, S1 S2 becomes the empty set. Looking at the definition of effectiveness of union, we see that P always contains S1 As2, and the effectiveness becomes greater. The end of the last example was a case in point.

Difference, project, and generalised project can all involve existential quantification. The evaluation mechanism for this would involve supplying the values for the unquantified selectors, and then applying the relation over which quantification is made to test for the existence of a member.

Ex. 5.3: to illustrate how quantification would be handled, let us take a very simple example. Let us find the set of numbers sx for which there is angle x such that sin(x)=sx. We will use the relation SINE from examples 4.6 and 4.8. The set that we want is obtained by projection, and is SINE │{sx} , with selector sx.

Suppose that SINE has effectiveness {{sx}, {sx, x}}. If we are given a value of sx, we can apply SINE to this to find a value of x, if any. If we find one or more values of x, then we know that one exists and the value of sx is in the set; and if no values of x are found, then sx is not in the set. Clearly then SINE│{sx} is effective as a recogniser, and we are using an effectiveness

El(SINE│{sx}) = {{sx}}.

However, suppose that the effectiveness of SINEis{{ x},{x, sx} }. We are helpless unless a value of x can be obtained from somewhere. It cannot be input, and the only way to obtain a value would be to use a representation of the set underlying the selector x. Possibilities in this direction are taken up in the next section. Without using the underlying set, the representation is ineffective, and thus

E2(SINE│{sx}) = {}.

If SINE was stored explicitly as a table, then it would have effectiveness {ϕ, {x}, {sx}, {x, sx}} , in which case no input would be necessary, and we could directly generate the set required. The effectiveness would be

E3(SINE│{sx}) = {ϕ, {sx}}.

In all cases these formulae are obtained directly from the formulae given above. □

As explained above, our interest in the effectiveness of relational expressions centres on whether or not they represent relations which are generators. The test for this is whether E(expression) contains the empty set. The process which computes the effectiveness of an expression should also record the way this was obtained, and thus record the particular evaluation process whereby the relation can be generated.

Relations as mappings from one of their components to all the components when the relations are stored explicitly as sets of tuples, can be viewed as ‘access paths’ in the sense of Earley [7]. Note, however, that here we have been considerably more general in that all the data need not be stored explicitly. With access paths in complex expressions one finds that many combinations of paths will Yield the same result. A small extension of our approach in computing the effectiveness of an expression will develop particular choices of access paths, and given that different access paths have

different speeds, a direct application of dynamic programming principles will select the best combination of access paths. A problem similar to this has been tackled by Stocker and Dearnley [13], but without the theoretical foundations. Note that such selections of best access paths can be very tricky, as has been pointed out by Hall [8] in the context of optimization.

Two results that can be obtained with our theory are interesting. These are stated below, without proof.

Proposition 5.1: if a relational expression represents a generator, then at least one operand of the expression is also a generator.
□

Proposition 5.2: Relational expressions that are equivalent within the algebra have the same effectiveness.
□

So far we have ignored the problem of infinite or potentially infinite relations. If we allow the generation of infinite (or indeed, very large) relations, we encounter three problems. One is the impossibility of displaying explicitly infinite sets, The others are more subtle. The result of a complete expression can be finite, even though some intermediate result is infinite; if we were to generate fully the intermediate relation, we would never terminate. In using a generator as a recogniser, we must be able to terminate in the case that the element is not in the set.

The termination problem needs solving. The tuple at a time evaluation mechanism gives a better solution then the complete evaluation of intermediate results, but this is not enough. Consider for example, the intersection of the countably infinite relation INT with the relation SMALL of selector x, which is a pure recogniser for numbers less than 100. INT*SMALL is then finite, containing all the integers less than 100, but evaluation of INT*SMALL would have to continue generating integers using INT ad infinitum just in case SMALL accepts another one. We must exploit further information in order to be able to terminate the generation of INT. The solution that we offer is to postulate orderings, either partial or total, on all underlying sets. However these orderings must be reasonable, in the sense that they will be useful in the solution to our problem. Not all orderings will be acceptable. For example, for strings the usual lexicographic ordering would be unreasonable, while the interpretation of strings as integers base r, where r is the number of symbols, is reasonable. These orderings then induce partial orderings on all relations, Now to control the termination of the evaluation of relational expressions, we propose to use supremum-infimum pairs as bounds for relations. We can then test an expression to ascertain the bounds for all the intermediate results, given the bounds for the operands. The method is obvious. If all generations proceed monotonically with respect to their respective orderings, then the bounds can be used to enforce termination upon potentially infinite relations.

The conversion of an generator for an infinite set into a terminating recogniser is a classical problem, and is solved by the method above, using orderings of the sets to detect termination in the case of non membership.

6. Using the Underlying Sets

In the discussion above we have assumed that the objects came from a single underlying set U. In discussing the effectiveness of the representation of relations, we

have not assumed anything concerning the representations of the underlying sets. However, it is well known from recursive set theory that if the underlying set is represented by a generator, then any recogniser of a set can be converted to a generator by generating successively potential members of the set and then testing for membership before releasing the element. This is equally true for relations, but in order to discuss this adequately we will suppose that we have a collection of underlying sets

$\Omega = \{Ui: i \text{ in some index set}\}$

The single set of objects U treated so far is simply the union of these underlying sets Ui. We now associate a single underlying set with each selector, with a mapping

$d:\Sigma \rightarrow \Omega$

For each selector we will talk of the effectiveness of the representations of the associated underlying set, writing E(d(s)) for selector s. Now we note that representations of some underlying sets are necessarily only recognisers (for example, the real numbers and all other uncountable infinite sets), while other underlying sets may be represented by either recognisers or generators. We express the effectiveness of a representation for the underlying sets in exactly the same way as before, writing E(d(s)) either as {{s}} or{ϕ , { s}} as appropriate. We will not consider the case where the underlying set is not effectively represented, with E(d(s)) = {}.

Cartesian products of the underlying sets would still be written as U(S), but now these mean something much smaller, since each selector is associated with a subset of universe of objects, U. This in turn affects the definitions of the relational operations, definitions 3.1 to 3.5.

The representations of the U(S) are obtained from the representations of the individual sets. Again the effectiveness of these representations would be expressed as a set of input sets, and thus the underlying sets can now appear in relational expressions. To convert all representations as near to generators as possible, we then simply combine relations with the cartesian product underlying them, using generalised intersection.

Ex. 6.1: we extend Ex. 4.6 to exploit the representation of the set underlying the selector x if this is possible. We have

SINE' = SINE * U({x, sx})

If the set underlying x is the real numbers, then we necessarily have

El(d(x)) = { {x} },

a pure recogniser for the reals. Substituting this in the formula for the effectiveness of the representation of SINE', we have

El(SINE') = {{x} , {x, sx}},

which is the effectiveness we had in Ex. 4.6. However, if the set underlying selector x is the rationals, represented by a generator, we find:

E2(d(x)) ={ϕ, {x}},

E2(SINE') ={ϕ , {x} {ss}, {x, sx}}.

The representation of the relation SINE' has become a pure generator as a result of exploiting the representation of the underlying set.

Underlying sets can also be exploited within relational operations. This is important within generalised union, because, as we saw, if underlying sets are not exploited, then in general only a recogniser results. The general strategy is to combine the underlying sets into the expressions using generalised intersection. Let us look at the Ex. 5.2 again.

Ex. 6.2. Using the relations SINE and COSINE, as in Ex. 5.2, we now exploit the underlying sets. Given a value for x, we can now apply SINE to this x to obtain the sx values, and then use the representation of d(cx) to generate values for the remaining component, and we can apply COSINE to x and then fill in the third component using d(sx). The relational expression that we are in effect exploiting is

(U(cx)*SINE)+(U(sx)*COSINE)

which has effectiveness { {x,} {x, sx}, {x, cx}, {x, sx, cx}}.
The fullest effectiveness that we can achieve is given by

U(x)*U(sx)*U(cx)*(SINE+COSINE)

which is a generator. This method for generating the tuples of the relation generates the tuples of the underlying cartesian product, and then uses the SINE+COSINE as a recogniser to select only those tuples that we want.

Note that distributing the U(si)'s inside the union gives us an effectiveness superficially like that we would have obtained had we exploited the underlying sets in the representations of the operands, but we do obtain extra power from using the underlying sets at the union. Clearly an important facet of the use of underlying sets is deciding just where to use their representations most effectively, without overdoing it and ending up with trivial and inefficient evaluation strategies.

Similar use of the underlying sets is possible for the other operations, and could be very useful in handling quantifications.

7. Conclusions and Open Problems

The theory that we have presented here forms the basis for the abstract syntax and semantics for a relational data base system. The extension of the language to include the usual arithmetic operations and comparisons within the concrete syntax are now trivial. For example, the relation SMALL of numbers less than 100 could be directly written as (x<100), while the relation which takes two values x and y to produce a third, z, which is their sum, could be written in the usual assignment notation, (z:=x+y). The effectiveness of these could be deduced directly from the surface syntax, and we might have, for example, the effectiveness of the adding relation as {{x, y}, {x, y, z}}. In this last, if we permitted the usual algebraic manipulations, a stronger effectiveness could be deduced.

HD23 Unfortunately the text doesn't clearly spell out that x<100 might be given as the condition in the "selection" operator that became : in ISBL (mistyped as ; in PRTV), SELECT in BS12, and WHERE in **Tutorial D**. Nor does it go so far as to suggest that an assignment such as z:=x+y might be given in connection with what in 1975 would have been a new relational operator—the # that was added to ISBL in March, 1976. It became CALCULATE in BS12 and EXTEND in **Tutorial D,** and is generally referred to as extension now. It is interesting to note that even those few early textbooks that give any information on ISBL fail to mention its operator named # (references [17] and [18], for example).

Consider a **Tutorial D** expression of the general form

EXTEND *r* : { *y* := *f*(*x*) }

and let *s* be the result of its evaluation. If we remove the colon and braces from the second operand, we have the predicate expression *y* = *f*(*x*), which

could be taken in this context as denoting the relation that represents the extension of that predicate. In that case, the expression is equivalent to *r* JOIN (*y* = *f*(*x*)). If we further take *x* as denoting an *n*-tuple (*n*⩾0), we can note that EXTEND requires the heading of *r* to be a superset of that of *x*. The implied join operation is normally expected to be "lossless" (i.e., the cardinalities of *r* and *s* are equal), but in ISBL that requirement was dropped, such that where *f*(*x*) is undefined (as in division by zero, for example) for some tuple of *r*, no corresponding tuple appears in *s*.

Note that as well as being convenient, EXTEND automatically limits the user to write only what HHT calls effective expressions—ones that are guaranteed to yield finite relations even when an operand might be infinite. Although it also requires *f*(*x*) to be a function, that particular limitation has been noted as theoretically unnecessary. It could perhaps also allow the use of non-functional operators such as "square root of", where a domain element can map to *n* range elements (where *n* is finite).

When *f*(*x*) itself is a predicate expression it can be used as a selection condition (in **Tutorial D**, *r* WHERE *f*(*x*)) and again we can regard it as denoting the relation representing the extension of that predicate, to be joined with *r*. Again, the heading of *r* has to be a superset of that of *x*, so again we can say that WHERE limits the user to write only effective expressions.

In recognition of the fact that restriction and extension can both be regarded as special cases of join, ISBL eventually allowed "filters" to be included in invocations of #, its extension operator. Thus, an expression of the form "*r* # *y*:=*f*(*x*), *y*<7, *z*=*g*(*y*)" was equivalent to "*r* # *y*:=*f*(*x*) : *y*<7 # *z*=*g*(*y*)", where : is restriction. And as already noted, even *y*:=*f*(*x*) could act as a filter in the case where *f*(*x*) is not defined for every tuple of *r*.

We are, however, left with some problems. The generalised union suggests the use of semi-procedural representations, placing a '*' in the positions where arbitrary elements of the underlying set would be placed. This then gives a finite representation of infinite sets. The symbol '*' would play a 'matches anything' role in the manner that is traditional in computing practice, but here we appear to have a firm theoretical underpinning. This has yet to be explored.

The evaluation mechanism has been presented informally. This needs a precise formulation for example by using the techniques of Hitchcock [9]. This would enable both the derivation of effectiveness from more primitive notions, and the application of termination theorems such as those of Hitchcock and Park [10]'

The characterisation of effectiveness is not the most powerful that we could use, and we could imagine partial representations which given a set of input values could compute some of, but not all of, the remaining components. We are developing a formalism to cope with this, basing it on the propositional calculus.

Details of the algorithms for obtaining the most effective representation for an expression have not been filled in. An associated problem concerned with the estimation of the cardinality of relations is still open.

8. Acknowledgements.

The train of investigations which led to this paper began with a Technical Note by Stephen Todd, concerning how to implement first order functions on relations within

the current experimental relational data base system at the IBM (UK) Scientific Centre, Peterlee, England. The current system is based upon the traditional relational algebra, and it soon became clear that a complete rethink of this approach was necessary. During the course of the investigations we had numerous discussions with colleagues at the Scientific Centre.

In particular, we acknowledge ideas concerning role names and how to model these which came from Ken Hanford (selectors in the spirit of ULD III [11], and Peter Quarendon (free variables and relations as predicates). We also thank John Owlett for discussions which helped clarify our thoughts.

9. References

1. Boyce, R.F., Chamberlain, D.D., King, W.F. III, Hammer, M.M.: 'Specifying Queries as Relational Expressions: SQUARE', IBM Research Report, RJ1291, Oct 16 1973.
2. Boyce, R.F., Chamberlain, D.D.: 'A Structured English Query Language', IBM Research Report, RJ1318, March 1974.
3. Bracchi, G., Fedeli, A., Paolini, P.: 'A Language for a Relational Data Base Management Systems', Proceedings of the Sixth Annual Princeton Conference on Information Sciences and Svstems, 1972.
4. Codd, E.F.: 'A Relational Model of Data for Large Shared Data Banks', Comm. ACM 13, June 1970, pp. 377-387.
5. Codd, E.F.: 'Relational Completeness of Data Base Sublanguages'. in 'Data Base Svstems', Editor Rustin, Prentice-Hall, 1972.
6. Codd, E.F.: 'Further Normalization of the Data Base Relational Model', in 'Data Base Systems', Editor Rustin, Prentice-Hall 1972.
7. Earley, J.: 'Relational Level Data Structures for Programming Languages'. Acts Informatica 2, 1973, pp. 293-309.
8. Hall, P.A.V.: 'Common Subexpression Identification in General Algebraic Systems', IBM UKSC Report UKSCO060, August 1974.
9. Hitchcock, P.: 'Fundamental Operations on Relations' , IBM UKSC Report, UKSCO051, May 1974.
10. Hitchcock, P. and Park, D.M.R.: 'Induction Rules and Termination Proofs', in Automata, Languages, and Programming, Editor M. Nivat, North–Holland/Elsevier, 1973, pp. 225-251.
11. Lucas, P., Lauer, P,, Stigleitner, H., 'Method and Notation for the Formal Definition of Programming Languages', IBM Lab., Vienna, Tech. Rep., TR25.087, June 1968.
12. Rissanen, J., Delobel, C.: 'Decomposition of Files, a basis for data storage and retrieval', IBM Research Report RJ 1220, May 1973.
13. Stocker, P.M., Dearnley, P.A.: 'Self-organising Data Management Systems', The Computer Journal, Vol. 16, no. 2, pp. 100-105.

The following are referenced in Hugh Darwen's annotations:

14 *The Third Manifesto* and much related literature is available at http://www.thethirdmanifesto.com.

15. Hugh Darwen's presentation slides and accompanying notes are available at https://www.northumbria.ac.uk/sd/academic/ee/work/research/computerscience/computational_intelligence/third_manifesto/.
16. The definitions of **Tutorial D** and **A** are available at http://www.thethirdmanifesto.com, in the section Reference Material.
17. David Maier: *The Theory of Relational Databases* (1983), available at web.cecs.pdx.edu/~maier/TheoryBook/TRD.html.
18 Jeffrey D. Ullman: *Principles of Database Systems* (2/e, Computer Science Press inc., 1982).
19 E.F. Codd: *The Relational Model for Database Management, Version 2* (Addison-Wesley, 1990).