McGill University

School of Computer Science
Burnside Hall (514) 392-8275

Aldat

Extending the Relational Algebra to Capture Less Meaning

T. H. Merrett
McGill University
Montreal

We argue that systems of operations on data are most effective when they are *formalisms*, in which semantic considerations are unimportant until the formalism is applied to some specific application. In this way, database processing can join the ranks of successful mathematical abstractions. Differential equations, for instance, can be applied to situations ranging from orbit calculations to the quantum mechanics of the atom. The semantics of each application is unique to that application, but the formalism of differential equations is common. The power of the formalism lies in its abstraction from issues of meaning.

This paper explores extensions to the relational algebra, made with practicality in mind, but under the constraint of generality, i.e., freedom from being limited to particular interpretations. By going to the foundations of the relational model in mathematical set theory we can generalize the now classical operations of projection, restriction, natural join and division to two useful families of binary operations and a flexible unary operator which adds quantifiers to the usual process of selecting tuples and attributes from a relation. By using the principle of algebraic closure, we can integrate arithmetical operations on the values of attributes in tuples into the algebraic framework.

We also discuss some of the areas of application which we have used as touchstones of practicality in our research: commercial and library information systems, text and graphics database processing.

1. Operations on Relations

The relational model is grounded in set theory - a relation is a set of tuples, or, more precisely, it is a subset of the cartesian product of its domains (the sets from which its attributes are drawn). A set is a special case of a relation, namely a unary relation. It is appropriate that operations on relations should generalize the operations already provided for sets. There are three classes of such operations: unary operations (complementation), binary operations which result in sets (union, intersection,etc.) and binary operations which result in logical or boolean values (inclusion, disjointness, etc.)

The unary operation of complementation does not seem to be very useful for relations, which are stored explicitly in a computer and which are usually sparse relative to the universe of all possible tuples. Complementation has not been used in queries or data processing because the notion of the universe is not free from ambiguity in the user's mind: we shall say more on this when we discuss quantification. It is the identity operation on sets which generalizes to the relational operations of project and restrict. (Sorting is such an identity operation, since the abstraction which created the notion of set ignores the order of the elements.) The binary set operations give easier generalizations, and we discuss them first.

1.1 Binary Operations

u-joins Both categories of binary operation on sets - those which produce sets and those which produce booleans - extend to relational operators which produce relations. We start with the first, the operations of union, intersection, difference and symmetric difference, and look at the natural join [Codd, 1970] in their light. If we specialize the natural join to operate on sets, we have exactly set intersection. We thus call natural join *intersection join*. There are conceivably many ways to generalize set union to relations, but it seems appropriate to generalize it in the same way that intersection extends to natural join. This has been done by a number of people, to give the *outer join* - see Date [1983] for some references. Define the *center* to be the natural join of relations R(X, Y) and S(Y, Z), here written R ∩ S. Define the *left wing* to be those tuples of R which do not participate in R ∩ S, augmented by null values for the attribute Z. Define the *right wing* similarly as the tuples of S which match no tuples in R, augmented by null values for attribute X. The *union join* (outer join) is just the set union of left wing, center and right wing. It specializes to set union. Similarly, we can extend the other set operations, and add the *left join*, which specializes to an identity operation.

$R \cap S =$ *center* (natural join)

$R \cup S \triangleq$ *left wing* ∪ *center* ∪ *right wing* (outer join)

$R + S \triangleq$ *left wing* ∪ *right wing*

$R - S \triangleq$ *left wing* (here it is convenient to project the *left wing* on attributes X and Y)

R *left* S $\triangleq$ *left wing* ∪ *center*

This family of joins, generalizing the set-valued binary operations on sets, is called the *u-join*.

σ-joins Relational division, presented by Codd [1971] as an algebraic counterpart to the universal quantifier, is a little hard for many people to understand. It becomes clearer when perceived as a generalization of set inclusion. For relations R(X, Y) and S(Y), the division $R \supseteq S$ is just those values of X which are associated by R with a set of values of Y which contain the set S, $R \supseteq S \triangleq \{x | R_x \supseteq S\}$ where R_x is the set of Y-values associated with x in R. This can be extended by allowing any set comparison, such as $\supset, \not\supseteq, =$ etc., to replace $\supseteq$. It can be further extended to relations R(X, Y) and S(Y, Z) by defining $R \supseteq S \triangleq \{(x,z) | R_x \supseteq S_z\}$ with S_z having a meaning similar to R_x. For completeness, we must introduce two new set comparisons and their complements: $\Cap$ tests the two sets compared for empty intersection and $\Cup$ tests whether they span the universe. The operation $R \not\Cap S$ is the *natural composition* [Codd, 1970], a most useful operation in its own right, although definable as a projection of the intersection join, but thereby seen to be a closer cousin to division than to the natural join.

This family of joins is called the *σ-join*. If we allow a relation with no attributes to be a *scalar*, taking on in this case the boolean values true or false, we see that the σ-joins specialize to the corresponding set comparison operators.

1.2 Unary Operations

QT-selectors The unary relational operators of *projection* and *selection* can be conveniently combined into a single operator with a simple syntax; for instance,

W, Y where (X < "tim" or Y = 2) in R

is a selection (X < "tim" or Y = 2) on a relation R(W,X,Y,...) followed by a projection on the attributes W and Y. The selection condition can be anything at all, provided it can be evaluated true or false on each tuple of the relation without reference to other tuples. It is called a T-condition (T for "tuple") for this reason, and the operation is a *T-selector*.

The *QT-selector* generalizes this to include *quantifiers* in the selection condition. The quantities envisaged are not limited to the classical ∃ ("for some") and ∀ ("for all"). The two quantifier symbols #("the number of") and • ("the proportion of") are used in quantifier expressions to generalize ∃ and ∀. Thus

W, Y where (# >1) Z, (X < 'tim' or Y = 2) in R,

applied to R (W,X,Y,Z,..), reads "find W and Y where, for at least one Z, X < 'tim' or Y = 2 in R"; and

X,Y where (•≤ .5) W, Z ≠ X in R

reads "find X and Y where, for no more than half of the values of W, Z ≠ X in R". The special cases ∃ and ∀ correspond to the quantifier expressions (# > 0) and (•=1), respectively. Quantifiers may be combined in one QT selector, and obey the rule followed by their set-theoretic special cases that changing the order of the quantifiers generally changes the meaning of the QT-selector.

While the QT-selector looks like a general query language facility, and exceeds most query langugages in functionality, it is only a unary operator on relations, satisfying the algebraic requirement of closure, namely that its result is in turn a relation. Thus QT-selectors may be nested or combined with other relational operators such as u-joins and σ-joins.

A word about the universal quantifier is advisable here. When we say "for no more than half the values of W", do we mean relative to all possible different values of W, relative to all values of W that appear in R, or something else? This is the problem of ambiguous - or just hard-to-calculate - universes. For various practical reasons, we define the universe in terms of the relation (or relational expression) appearing after the <u>in</u> keywork.

QT-selectors can be used to identify subsets of a relation to be updated, using a more extensive notation. We look next, however, at a simple operator permitting tuple-at-a-time inspection and updating of a relation.

<u>Relational Editor</u> The operations of the relational algebra discussed so far have obeyed two fundamental algebraic principles. The first is the principle of *closure*, that a relational operation produce a relation as a result. This permits the construction of relational expressions consisting of one or several operations. The second principle is *atomicity*, which permits us to ignore the internal structure of relations.

Thus we have been able to ignore the tuples which constitute the relations we have been operating on. Our notation has not even required tuple variables, range statements or *for each* loops, which explicitly focus attention on the microstructure of relations in many systems and notations. For the person responsible for the design of an information system, freedom from this distraction is ideal. This is the person who applies the formalism to a particular problem. We will call him the "programmer user".

For the person who actually uses the data, however, the tuples are of primary interest and the fact that they are grouped into relations is only incidental. This person enters the data and reads the reports. We will call her the "end user". One important activity for the end user is to examine and possibly change the tuples using an interactive editor. How can we reconcile the two complementary views the programmer user and the end user have of the data in a single operation?

Our answer is a two-faced operator. To the programmer user, the relational editor looks like a simple unary relational operator, like project for instance. A simple system such as <u>edit</u> R will result in a new relation (which may be assigned back to R). Only instead of the operation being defined algorithmically, as would be the case for the projection X,Y <u>in</u> R, the output relation is a result of the free activity of the end user at an interactive terminal using an appropriate command language. This interactive command language is the second face of the <u>edit</u> operator, the face presented to the end user, who can thereby see and edit the tuples of her data.

. . .

The foregoing is an outline of an extended relational algebra which is discussed precisely and more fully by Merrett [1983]. A system including all the above operations has been implemented in U.C.S.D. Pascal on an Apple II [Chiu, 1982].

2. Operations on Attributes

Many practical problems for databases require computations to be performed on attributes. For instance, multiplying an income by a tax rate for every tuple in a salary relation, or totalling sales by department. Most implemented systems handle at most some of the possible requirements, and these in an *ad hoc* fashion, such as by special functions for COUNT, TOTAL, AVERAGE, etc. The failure of these approaches to adopt a framework for operations on attributes which is consistent with the operations they provide for relations leads to dislocations of syntax and to incompleteness.

To be consistent with the relational algebra, we must embody our operations on attributes in an algebraic framework and in particular, we must observe the principles of closure and of atomicity. Closure requires that we operate on attributes to get attributes and atomicity requires that we avoid considering the structure of attributes or their connection with particular relations. The resulting formalism we call the *domain algebra*, and it comes in two flavours, horizontal and vertical.

2.1 Horizontal Operations

Horizontal operations are defined on any one tuple and are applied repeatedly to each tuple in a relation. They can be any arithmetical, logical or other operation or expression. They are "horizontal" because they can be imagined to operate horizontally along each row of the usual tabular representation of relations. In keeping with our algebraic principles, of course, there is no mention of tuples or of relations in the syntax:

```
            let MARK be  MIDTERM + FINAL + ASSIGNMENTS
```

creates an attribute MARK from the existing attributes MIDTERM, FINAL and ASSIGNMENTS.

Since MARK has been defined above in the absence of any particular data, it is a *virtual* attribute until it is *actualized* in the context of some particular relation. The most obvious mechanism for actualization is an operation of the relational algebra, such as projection. Thus, if we have a relation COURSE (STUDENT, SECTION, MIDTERM, FINAL, ASSIGNMENTS) we could project the relation:

```
            STUDENT, MARK in COURSE.
```

2.2 Vertical Operations

If horizontal operations work "along the tuples", vertical operations work "down the attributes", and fill the roles of totalling, subtotalling, integrating and partial integrating and their generalizations. For the first we have *reduction*:

```
            let TOTAL be red + of MARK
            let COUNT be red + of 1
            let AVMK be TOTAL/COUNT    ← a horizontal operation
            let MAXMK be red max of MARK
```

for the second we have *equivalence reduction*:

```
            let SUBTOT be equiv + of MARK by SECTION
            let SUBCT be equiv + of 1 by SECTION
            let SUBAVMK be SUBTOT/SUBCT    ← a horizontal operation
            let SUBMXMK be equiv max of MARK by SECTION
```

for the third and fourth we have *functional mapping* and *partial functional mapping*:

```
          let INTF be  fcn + of F order X
          let PINTF be par + of F order X by Y
```

We do not discuss these latter except to remark that, in the above simple forms, they provide poor integrals, but can be improved to perform as well as the data will allow. Note the ordering clause, which effectively specifies a sort on a given attribute or set of attributes, and makes very useful the special operations pred (predecessor in the ordering) and succ (successor in the ordering):

```
          let NEXTEVENT be fcn succ of EVENT order TIME.
```

While these vertical operations of the domain algebra were motivated by practical considerations arising from the application areas we have investigated in our research, fundamental principles and basic mathematical concepts have been used in their definition. Thus, apart from the principles of algebraic closure and algebraic atomicity, we have applied the notion of "equivalence class" in equivalence reduction and the notion of "functional" in functional mapping. Note that the order clause in functional and partial functional mapping does not violate the relational abstraction that order does not matter. The result of an operation actualizing one of these attributes is still a relation, and order does not matter. Functional mapping cannot, for instance, be used to print out a given relation in order: that must be done by a specialized print routine, which is not of particular interest to the relational algebra because it is at best an identity operation.

...

The foregoing is an outline of a domain algebra which is discussed precisely and more fully by Merrett [1983]. A system including this domain algebra and the relational algebra of the previous section has been implemented in U.C.S.D. Pascal on an IBM Personal Computer [Van Rossum, 1983].

3. Theoretical Implications

The theory of database design began as an attempt to put semantic considerations back into the relational model. The earliest study of the problems of constituting the relations making up the database for some particular application was Codd's [1971b] investigation of anomalies that arise when updating relations that contain certain functional dependencies. Functional dependence is a semantic constraint that can be imposed on the relational model in order to restore, for instance,the semantics that are inherent in the DBTG "network" model of data. The presence of functional dependence requires the database to be decomposed into relations in various ways if certain anomalies are to be avoided. The instruments of the decomposition are the algebraic operations of projection and natural join: projection to decompose and natural join to put the pieces back together again and confirm that no information has been lost. Extensive research followed from these beginnings, uncovering various new constraints, dependencies and their generalizations. All this work was based on projection and natural join as fundamental operations, with an occasional investigation of specialized selections and set union. This work has turned up perhaps more than its fair share of excessive complexity and even undecidability.

Possibly as a result of these difficulties, possibly for other reasons, design theory appears to even a sympathetic follower to have wandered far from its original source in formalizing and integrating semantic constraints into useful database designs in pratice. It copes with only limited semantics and makes no use of the generalized operators defined in the previous sections. Since these generalized operators are provided to extend the applicability and hence the semantic range of the relational algebra, it is plausible that much interesting theory could be based on them. Further details are beyond our competence and present purview, but it can be noted that while decomposition like that already investigated by theory may be potentially based on the μ-joins, the σ-joins do not lend themselves to this aspect of database designs.

4. Practical Justifications

While the above discussion has been couched in terms of principle and mathematical concept, each extension and generalization we have made to Codd's original relational algebra has resulted from a practical inadequacy of an existing system. We have investigated, in generalized and somewhat abstract form, commercial, library, geographical and text information systems. Commercial systems discussed in [Merrett, 1983] include manufacturing and financial information systems. We have investigated both administrative systems for libraries - e.g., acquisition and circulation - and information retrieval aspects of document clustering and search. Geographical databases have given us a framework for the study of information systems based on large two-dimensional images and diagrams. Text processing has been examined in a variety of ways, including editing and page formatting only as two of a broad range of processing: indexing, transliteration, linguistic analysis, encryptation, etc. The formalisms of the relational and domain algebras have been adequate to handle all the well-defined operations of these widely differing semantic contexts, although they strain a little around the intricacies of document classification and search, and special routines must be provided, resembling the relational editor, for the interactive operations of picture and text editing.

4.1 Financial Statements

To illustrate commercial and administrative information systems, which largely deal with formatted or tabular data, we can look at a simplified example of preparing funds flow and income statements from a spread sheet. The calculations shown here are typical of those required by information systems, which we have examined which process tabular data. The spread sheet is a form of double-entry bookkeeping which records all transactions between debit accounts (ACCTDB) and credit accounts (ACCTCR). We will consider only one time period, and assume that the spread sheet also contains descriptions of the transactions (TDESCR).

SPREAD (ACCTCR	ACCTDB	AMOUNT	TDESCR)
A	G	2.33	Assembly Cost
F	E	0.22	Fixed Asset Depreciation
M	A	1.30	Raw Materials Cost
P	E	2.11	Manufacturing Expenses
T	E	0.95	Payroll Taxes
E	A	1.43	Variable Costs
E	R	2.21	Gross Profit
.	.	.	.
.	.	.	.

(The example shows all transactions for accounts E and A, but not all for the other accounts: the full spread sheet must balance, of course).

The accounts are further described in a relation TYPE, and classified along traditional accounting lines into Asset, Liability or Equity accounts.

TYPE (	ACCT	ACTYPE	ADESCR)
	A	Asset	Goods in Assembly
	C	Asset	Cash
	E	Equity	Stockholders'
	P	Liability	Accounts Payable
	⋮	⋮	⋮

We will compute the Funds Flow statement, FUNDFLOW (ACCT, ADESCR, ACTYPE, FF) where FF is the net funds flowing into the account if it is an Asset account and flowing out otherwise. This is a summary of the spread sheet, and so involves subtotals, implemented by equivalence reduction. The union join is used to combine the totals of funds flowing in and funds flowing out to prevent losing information about any account that did not experience both an inflow and an outflow. TYPE is brought in by intersection join to provide ACTYPE and ADESCR.

```
let TOTIN be equiv + of AMOUNT by ACCTDB
let TOTOUT be equiv + of AMOUNT by ACCTCR
let FF be if ACTYPE = 'Asset' then TOTIN-TOTOUT else TOTOUT-TOTIN

FUNDFLOW ← ACCT, ADESCR, ACTYPE, FF in (
        (ACCTDB, TOTIN in SPREAD)  [ACCTDB ujoin ACCTCR]
        (ACCTCR, TOTOUT in SPREAD) [ACCTCR ijoin ACCT] TYPE)
```

FUNDFLOW	(ACCT	ADESCR	ACTYPE	FF)
	A	Goods in Assembly	Asset	0.40
	C	Cash	Asset	-5.58
	E	Stockholders'	Equity	0.36
	P	Accounts Payable	Liability	-4.58
	⋮	⋮	⋮	⋮

The income statement can be obtained from FUNDFLOW and REVENUE and EXPENSE, derived from SPREAD. The revenue is made up of those amounts credited to the stockholders' account, E, while the expenses are those amounts debited from E.

```
REVENUE ← TDESCR, AMOUNT where ACCTCR = 'E' in SPREAD
EXPENSE ← TDESCR, AMOUNT where ACCTDB = 'E' in SPREAD
```

In computing the income statement, we must combine revenues and expenses, retaining information about which entry is a revenue and which is an expense. We must also extract the net change in E from FUNDFLOW and record a profit or a loss according to whether the change is positive or not. We see some uses of the domain algebra which would be difficult in a less flexible formalism, including the generation of constant attributes.

The union join is used to combine REVENUE, EXPENSE and FUNDFLOW: here it plays the role of set union.

```
let REV be 'Revenue'; EXP be 'Expense'
let PL be if FF ≥ 0, then 'Profit' else 'Loss'
let PLDESCR be 'Net '||PL||' after taxes'    <<||concatenates strings>>
let PLAMT be abs(FF)

INCOME ← REVENUE [TDESCR, AMOUNT, REV ujoin, TDESCR, AMOUNT, EXP]
         EXPENSE [TDESCR, AMOUNT, EXP ujoin, PLDESCR, PLAMT, PL]
         (PLDESCR, PLAMT, PL where ACCT = 'E' in FUNDFLOW)
```

Here is INCOME displayed as a conventional income statement, as might be produced by a specialized procedure for printing the relation. The totals would be provided by suitable equivalence reductions.

EXPENSE		REVENUE	
Manufacturing Expenses	2.11	Gross Profit	2.21
Depreciation of Fixed Assets	0.22	Variable Costs	1.43
Payroll Taxes	0.95		
	3.28		
Net Profit after taxes	0.36		
	3.64		3.64

The calculation of financial statements from operational data is worked more fully as an illustration of algebraic techniques by Merrett [1983].

4.2 Text Processing

A very good reason for representing text as a relation is the variety of computations we can perform on it using the relational algebra. Text processing should not be limited to editing and formatting (page layout, typesetting). To represent text - which can in its simplest essence be seen as a sequence of words - in relational form, we must add a sequence number. Any sequence can be represented as a set by extending the set elements to include sequence numbers, despite the apparent differences between sequences and sets (order does not matter in a set; duplicates are allowed in a sequence). An encouraging consequence of this approach is that a *text is its own concordance*, if we take a concordance to be an index to all words in the text: as a text, the data is usually arranged in sequence number order; as a concordance it appears in alphabetical order of words. Here is a text, a list of "stop words" - insignificant but frequently occurring words - and an improved concordance, calculated by removing stop words using the difference join.

```
CONCORDANCE ← CORPUS djoin STOP
```

CORPUS (WORD	DOC	SEQ)
Little	A	1
Bo	A	2
Peep	A	3
has	A	4
:	:	:
Baa	B	1
baa	B	2
black	B	3
sheep	B	4
have	B	5
:	:	:

STOP (WORD)
a
alone
and
any
behind
come
does
for
has
have
he
:

CONCORDANCE (WORD	DOC	SEQ)
baa	B	1
baa	B	2
bags	B	14
black	B	3
bo	A	2
bowl	D	25
boy	B	28
called	D	17
called	D	21
called	D	28
cat	C	5
:	:	:

We can begin a process of clustering the different documents in CORPUS by calculating *term vectors* - a list of words and their relative frequencies - and a *similarity matrix* associating the documents. The term vectors hold normalized frequencies and the similarity coefficient used is the cosine coefficient or scalar product of the normalized frequencies. Note that we must find the natural join of TERMVECT with itself on WORD in order to calculate the contribution of each word to the cosine coefficient. The equivalence reduction used to find COSINE sums these contributions over all words common to each pair of documents. The assignment creating TERMVECT1 just copies TERMVECT, renaming attributes so the join can be done correctly.

```
let WORDFREQ be equiv + of 1 by WORD, DOC
let NORMFREQ be WORDFREQ/sqrt (red + of WORDFREQ↑2)
TERMVECT ← WORD, DOC, NORMFREQ in CONCORDANCE
```

TERMVECT(WORD	DOC	NORMFREQ)
baa	B	$2/\sqrt{20}$
bags	B	$1/\sqrt{20}$
black	B	$1/\sqrt{20}$
bo	A	$1/\sqrt{11}$
:	:	:

```
let COSINE be equiv + of NORMFREQ × NF1 by DOC, DOC1
TERMVECT1 [WORD, NF1, DOC1 ← WORD, NORMFREQ, DOC] TERMVECT
DOCSIM ← DOC, DOC1, COSINE where DOC ≠ DOC1 in
                    (TERMVECT ijoin TERMVECT1)
```

Here is DOCSIM in matrix form. Blank entries are zero and the matrix is symmetric.

DOC \ DOC1	A	B	C	D
A		.13	.07	
B	.13		.05	.04
C	.07	.05		
D		.04		

4.3 Geometrical Computations

A *diagram* is a line drawing, as used in maps, charts, graphs and technical illustrations. It is two dimensional and contains *points, lines,* and *regions* as elements. We will limit our attention to *polygonal* representations of diagrams, in which lines and the boundaries of regions are approximated by straight line segments. This representation has the advantages that polygons have been the recent subject of intense investigations in computational geometry and that they can be stored as *sequences* of points, so that all three types of element (point, line and region) can be represented uniformly.

A *feature* is a diagram or a subdiagram identified by a name, consisting of one or more points, lines or regions, i.e., *groups* of sequences. A relational representation of the Aldat logo, at the beginning of this paper, is shown in DIAGRAM. The feature Aldat is considered in this example to be a pair of closed lines: the assumption of the pred and succ operators defined in Section 2.2 is that ordering is cyclic, so that closed lines are easy to achieve. To leave a line open, the tuple connecting the highest and lowest sequence numbers of each group must be found and removed after any succ or pred operation. A region is always bounded by a closed line - either of the hexagons in Aldat could be considered a region. A point is a singleton sequence. Note that the representation in DIAGRAM does not cope with the hierarchy of picture, subpicture, sub-sub-picture, etc., which is important in many diagrams, but this is an easy extension - see Merrett [1983]. Note also that DIAGRAM is not in third normal form [Codd, 1971 a]: the presence or absence of this normalization is not of concern here.

DIAGRAM (FEATURE	TYPE	GROUP	SEQ	X	Y)
Aldat	line	1	1	-.5	-1
Aldat	line	1	2	.366	-.5
Aldat	line	1	3	.366	.5
Aldat	line	1	4	-.5	1
Aldat	line	1	5	-1.37	.5
Aldat	line	1	6	-1.37	-.5
Aldat	line	2	1	.5	-1
Aldat	line	2	2	-.366	-.5
Aldat	line	2	3	-.366	.5
Aldat	line	2	4	.5	1
Aldat	line	2	5	1.37	.5
Aldat	line	2	6	1.37	-.5

Some geometrical queries which could be made on DIAGRAM are the following. We show the algebraic operations to answer some of them. "Point" means any point already in the database. "Arbitrary point" means any point. Note that we assume, given a line, 'name', or a region, 'name', that this is the only line or region in the feature 'name'. Questions 7 and 8 raise some technicalities from computational geometry which we do not elaborate on here.

1. Find a given feature, 'name' (point, line or region)
   TYPE, GROUP, SEQ, X, Y where FEATURE = 'name' in DIAGRAM

2a. Given a point,(PX, PY), find what line it is in.
   FEATURE where (TYPE, X, Y) = ('line', PX, PY) in DIAGRAM

2b. Given a line, 'name', find what points are in it.
   X,Y where (FEATURE, TYPE) = ('name', 'line') in DIAGRAM

2c. Given a region, 'name', find its boundary.
    SEQ, X, Y <u>where</u> (FEATURE, TYPE) = ('name', 'region') <u>in</u> DIAGRAM

2d. Given a closed line, find the region it bounds.

3a. Given an arbitrary point, (PX, PY), find what region it is in.

3b. Given a line, find what region it is in.

3c. Given a region, find what points are in it.

3d. Given a region, find what lines are in it.

4a. Given two lines, 'name 1' and 'name 2', find their intersection.

4b. Given two regions, find their intersection.

5a. Given a line and two arbitrary points on it, find the length of the line between the points.

5b. Given a closed line bounding a region, find the area and centroid of the region.

6a. Given an arbitary point, find the nearest point.

6b. Given an arbitrary point, find the nearest line.

6c. Given a line, find the nearest different line.

6d. Given a line, find the nearest point not on it.

7. Given a region, find its convex hull, visibility graph, medial axes, triangulation...

8. Given a set of points, find the Voronoi diagram and the Delauney triangulation.

We elaborate briefly on queries 3a and 4a. We use the sum of angles method for 3a, which says that angles of rays drawn from a point to all vertices of a polygon sum to $\pm 2\Pi$ if the polygon contains the point, and to 0 otherwise. The angle is calculated using $C^2 = A^2 + B^2 - 2AB \cos \Theta$ where C is the length of the subtended side of the polygon and A and B are the lengths of the adjacent rays. This does not give the sign of θ, the angle, which is positive or negative depending on whether the order of the points (X,Y), (X',Y') and (PX, PY) is counterclockwise or clockwise, where (X',Y') is the cyclic successor of (X,Y). This is given by the sign of the determinant of the 3 × 3 matrix

$$\begin{pmatrix} 1 & X & Y \\ 1 & X' & Y' \\ 1 & PX & PY \end{pmatrix}$$

```
let A2 be (X-PX)↑2 + (Y-PY)↑2
let X' be par succ of X order SEQ by GROUP
let Y' be par succ of Y order SEQ by GROUP
let B2 be par succ of A2 order SEQ by GROUP
let C2 be (X-X')↑2 + (Y-Y')↑2
let ANGLE be arccos ((A2 + B2 - C2)/(2 × sqrt(A2 × B2)))
let AREA be det3(1,X,Y,1,X',Y',1,PX,PY)
let TOTANG be equiv + of  sign(AREA) × ANGLE by FEATURE, GROUP
FEATURE, GROUP where abs(TOTANG) = 2 × Π and TYPE = 'region' in DIAGRAM
```

A test with point ($-\frac{1}{2}$,0) against the Aldat logo (with the hexagons considered regions not lines) shows that the point is contained in the first group (hexagon) but not the second. Point (0,0) is in both.

In query 4a, we must combine every edge of the first line with every edge of the second line. This involves selecting the lines and finding their cartesian product. Note how we use <u>ijoin</u> for this. Edges, of course, are determined by pairs of points generated by the successor operation.

Once all combinations are found, we can compute the intersection coordinates for each combination and determine whether they lie within the base defined by the four endpoints of the two intersecting edges. We assign the result coordinates the "don't care" null values, *DC*[see Merrett, 1983], if the lines are parallel.

```
let X' be par succ of X order SEQ by GROUP
let Y' be par succ of Y order SEQ by GROUP
let XX be X; YY be Y; XX' be X'; YY' be Y'
let A be X-X'; B be Y'-Y; C be Y × A + X × B
let AA be A; BB be B; CC be C
let DEN be B × AA - BB × A
let YP be if DEN = 0 then DC else (B × CC -BB × C)/DEN
let XP be if DEN = 0 then DC else if B = 0 then (CC - AA × YP)/BB else
                                              (C-A × YP)/B

XP,YP where X min X' ≤ XP ≤ X max X' and
            XX min XX' ≤ XP ≤ XX max XX' and
            Y min Y' ≤ YP ≤ Y max Y' and
            YY min YY' ≤ YP ≤ YY max YY' in (
  (X,X',Y,Y',A,B,C where (FEATURE, TYPE) = ('name 1','line') in DIAGRAM) ijoin
  (XX,XX',YY,YY',AA,BB,CC where (FEATURE, TYPE) = ('name 2','line') in DIAGRAM))
```

This code works for closed lines, such as the hexagons in 'Aldat'; they intersect at points (0,-.711) and (0,.711). For open lines, we must add to the selection conditions before the ijoin the proviso that SEQ not exceed its successor.

While pursuing the polygonal approach to geometrical data, we have not mentioned its major rival, the grid or pixel approach. A rectilinear grid of picture elements ("pixels"), each with a variable grey level (or colour level), is very suitable for wirephotos, television images, landsat data, etc. It is less flexible for processing but has the advantage of being able to localize interesting parts of the picture very well because of the simple cartesian coordinate system it provides. Our research into storage structures for relations has produced multipaging [Merrett, 1983], which permits a hybrid of polygon and grid data structures for diagrams, with the best features of both representations [Düchting, 1983].

5. Semantic Restrictions

The preceeding examples show that we can come a long way on pure formalism. A sufficiently general formalism lends itself to semantic interpretation in many different ways, but is always subject to the same rules of manipulation, independent of interpretation. In the end, of course, we succeed in capturing "more meaning", as is the goal of other workers in this field. However, our approach is different, being to abstract, generalize and formalize rather than to seek out and elaborate particular patterns of meaning.

One area where this program has so far met with difficulty has been the operation of transitive closure and its kindred. For instance, analyzing PERT networks or a Bill-of-Materials are closely related processes. It would be nice to have a transitive closure operation, but we have not seen how to generalize it interestingly to arbitrary relations. Transitive closure and its generalizations are applicable only to relations whose key is a pair of attributes drawn from the same domain - a relation with the "topology" of a graph.

This leads us to suggest semantic *restrictions* - subsets of all possible relations with additional well-defined properties, which also have additional operations not generally applicable to all relations. An example is the topological graphs just mentioned, with the operation (among others) of transitive closure: we can also search, find spanning trees, connectivity, minimal-cost paths, etc. A second example is the class of relations which embodies *sequences*, as we have already discussed in text and graphics, above. Operations include insertion and deletion of elements, and a family of *sequence merges* which resemble the μ-joins on relations, only it results in the union, intersection or difference of polygons when the sequences are so interpreted. Another interpretation of a sequence could be *time*, as in history or chronological relations.

It may seem a pity to have to restrict relations to these semantic classes for the definition of special operators, but if a high level of abstraction and formalization is kept in these classes and, above all, if the full underlying formalism of the relational algebra is preserved, investigation is bound to be fruitful.

Acknowledgements

I thank the Natural Science and Engineering Research Council of Canada and the Fonds formation de chercheurs et actions concertêes du Quêbec for support under grants NSERC A4365 and FCAC EQ2002 respectively. Debbie Morris and Vera Gorni typed this paper.

References

G.K.-Chiu, 1982. MRDSA User's Manual, McGill University School of Computer Science, Tech. Rept. SOCS-82-9 (May, 1982).

E.F.Codd, 1970. A relational model of data for large shared data banks. CACM 13 6(June, 1970) 377-87.

E.F.Codd, 1971a. Further normalization of the data base relational model in R. Rustin, ed. Data Base Systems, Prentice Hall, Engelwood Cliffs, N.J. (1972) 34-64.

E.F.Codd, 1971b. Relational completeness of data base sublanguages, *ibid*, 65-98.

C.J.Date, 1983. The outer join in Proc. 2nd Internat. Conf. on Databases (ICOD-2), S.M.Deen & P. Hammersley, eds., Cambridge, U.K. (Sept. 1983), 76-106.

B. Düchting, 1983. A relational picture editor. McGill University School of Computer Science Tech. Rept. SOCS-83-19 (Aug. 1983).

T.H.Merrett, 1983. Relational Information Systems. Reston Publishing Co., Reston, Va.

T.Van Rossum, 1983. Implementation of a domain algebra and a functional syntax for a relational database system. McGill University School of Computer Science Tech. Rept. SOCS-83-18 (Aug.1983)