BASICS: Relational Algebra
Relational Information Systems Chapter 2.1
(Revised 99/9)

November 30, 1999

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

The use we advocate for relations in systems using secondary storage is as the primitive units of data, just as integers and reals are the primitive units of data in a numerical calculation in RAM. (This is not to preclude using relations for calculations in RAM as well, when they apply.) It is not adequate simply to have a way of describing data: we also need a way to manipulate it. The relational approach is the best of the major data models for our purpose because it gives us not only the relational form for data but also the *relational algebra* to process data with.

The relational algebra is the first of two major systems of operations that we will discuss, which combine to make up the basis of the *al*gebraic *dat*a language, Aldat. Figure 1 outlines a taxonomy which we shall elaborate on in this and succeeding chapters.

The first essence of an algebraic approach to manipulating relations is that relations are considered as atomic constructs by the operation. Thus, access to tuples within a relation is precluded: Aldat has no notion of tuple. This greatly simplifies the ntation and manipulations that must be done. It may seem unduly restrictive: we aim to demonstrate, however, that an appropriate selection of relational operations is remarkably flexible in an area of applications such as information systems.

We even elevate the foregoing into a *Principle of Abstraction*: *the structure and the context of a relation should be of no concern to the operation.*

The second essential aspect of the algebraic approach is that the set of relations is closed under the operations. Operating on relations gives new relations. This makes possible the construction of expressions of arbitrary length and complexity, just as numerical expressions can be built up of numerical operators and a closed set of numbers.

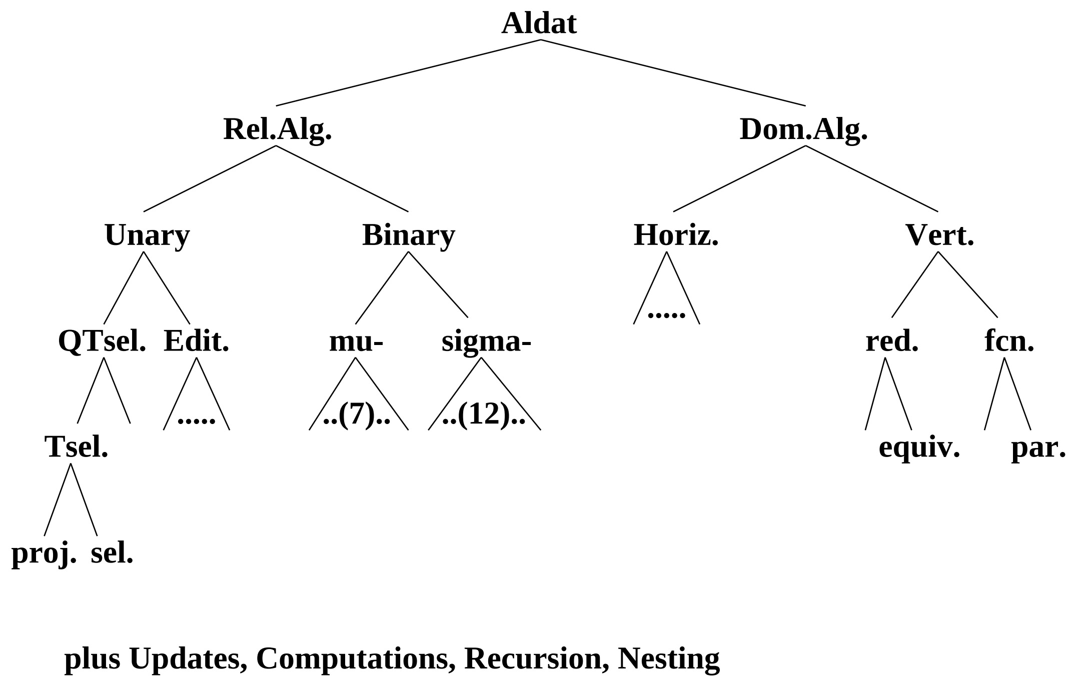


Figure 1: A Taxonomy of Aldat

This gives the *Principle of Closure*: *operations on relations should produce relations.*

An example of violation of the principle of closure would be a system which operates on relation to produce displays or printouts instead of relations which can be further manipulated.

1 Assignments and Views

It is useful to be able to create new relations from old ones, and we start with a notation for assignment operators, which assigns a value to a relation. Almost all the rest of the relational algebra discussed in this chapter is "functional": there are no "side effects" such as would be caused by assignment or by updating. But once we have written a functional relational expression, it is important to be able to tell the system to execute the expression and to put the result somewhere.

Figure 2 shows four types of assignment operator. The *replacement* operators completely replace the left-hand operand, which need not be previously defined, or which can have been previously defined on completely different attributes from those that will result from the assignment: the old definition is destroyed, and all the old data. The *incremental* operators add new tuples, and the attributes of the relation on the right must be compatible with those of the relation on the left of the assignment. The *renaming* assignment allows attributes on the left to be matched with attributes on the right: all attributes of the left-hand relation must be specified in the list.

While the assignment operator causes the expression following it to be evaluated and the result stored in the relation named on the left, it is useful to be able to defer the evaluation until later. The mechanism for this is called a *view* in databases, and a function (without

		Renaming
Replacement	$T <\!\!- R$	$T[B,C,A <\!\!- A,D,E]S$
Incremental	$T <\!\!+ R$	$T[B,C,A <\!\!+ A,D,E]S$

Initial values:
relation $R(A,B,C)$ <– {("a","b","c")};
relation $S(A,D,E)$ <– {("w","d","e")};
Sequence of assignments:

Assignment	Result $T(A,$	$B,$	$C)$
$T <\!\!- R$	a	b	c
$T[B,C,A <\!\!- A,D,E]S$	e	w	d
$T <\!\!+ R$	a	b	c
	e	w	d
$T[B,C,A <\!\!+ D,E,A]S$	a	b	c
	w	d	e
	e	w	d

Figure 2: Four Types of Assignment

Responsibility

(*Agent*	*Item*)
Raman	Micro
Raman	Terminal
Smith	V.C.R.
Hung	Micro

[*Item*] **in** *Responsibility*

(*Item*)
Micro
Terminal
V.C.R.

Figure 3: Unary Operators of the Relational Algebra: Project

parameters) in programming languages. In our notation, **is** replaces the assignment arrows, <– and <+. Thus, T **is** R just defines T to be synonymous with R, and no evaluation is performed until a subsequent assignment, or other operation such as *print*, forces it.

All this becomes much more interesting when the right-hand side can be an expression of the relational algebra, involving relations and operations.

2 Taking Relations Apart—Unary Operations

Unary operations take a single operand, which is a single relation in the case of the relational algebra. The unary operations of the relational algebra stem from the original operations proposed by Codd [Cod70], only slightly generalized. Figure 3 shows *projection*, which creates a new relation of tuples on a specified subset of the attributes of the operand.

The second unary operation is *selection*, illustrated in figure 4. This selects tuples of a relation according to a Boolean condition on the values in the tuples. The Boolean may involve arbitrary operations on any attributes of the relation or on constant values, but must be able to be evaluated on each tuple independently of other tuples.

```
where  Item="Micro" in Responsibility
(Agent   Item)
Raman   Micro
Hung    Micro
```

Figure 4: Unary Operators of the Relational Algebra: Select

```
[Agent] where Item="Micro" in Responsibility
(Agent)
Raman
Hung
```

Figure 5: Unary Operators of the Relational Algebra: T-Selector

It is useful to combine select and project in a single operation, the *T-selector*. This is named to reflect the fact that tuples are selected according to their own values, independently of other tuples. The syntax shown in figure 5 is evaluated from right to left: the selection is done first, then the projection. This order of evaluation enables all the attributes of the operand relation to participate in the selection.

We can define T-selectors generally.

- For a relation, $R(X, Y)$, defined on disjoint sets of attributes, X and Y,
- $[X]$ **where** <cond(X, Y)> **in** $R \equiv \{x \mid (x, y) \in R \text{ and } \mathrm{cond}(x, y)\}$
- **where** <cond(X, Y)> **in** $R \equiv \{(x, y) \mid (x, y) \in R \text{ and } \mathrm{cond}(x, y)\}$
- $[X]$ **in** $R \equiv [X]$ **where true in** R

Generally, X is a non-empty set of attributes. In the special case that X is empty, the T-selector, $[X]$ **where** <cond(X, Y)> **in** $R \equiv [\,]$ **where** <cond(Y)> **in** R, is a Boolean, **true** if cond(Y) is satisfied by any tuple in R, **false** if not. $[\,]$ **where** <cond(Y)> **in** R can be read "there is something where cond(Y) in R", or "something where cond(Y) in R". See the end of section 3.2 for further motivation for this interpretation of *nullary* relations.

A second category of unary relational operator is the family of *editors*. This includes potential interfaces for graphics, spreadsheets, logic languages such as Prolog, array languages such as APL, etc. Here we look briefly at only a general relational editor.

To understand the approach taken here, we must distinguish between two classes of user. The *programmer user* is the person who writes code using the relational algebra. There is no concept of "tuple" in the relational algebra, in order to keep it abstract and at a high level: the only base concepts are relation and attribute. The programmer user is not interested in the detailed data, and so not concerned with the tuples that make up heir relation. The programmer user's perspective on editing should be that the editor is a unary operation, like project, which executes for a while then returns a relation.

The *end user*, on the other hand, is only interested in the tuples, and not concerned with how they may be aggregated into relations for the convenience of the programmer user. An

1. Programmer-user $R <- [X]$ **edit** R;
2. End-user tuple-at-a-time (*TATI*):

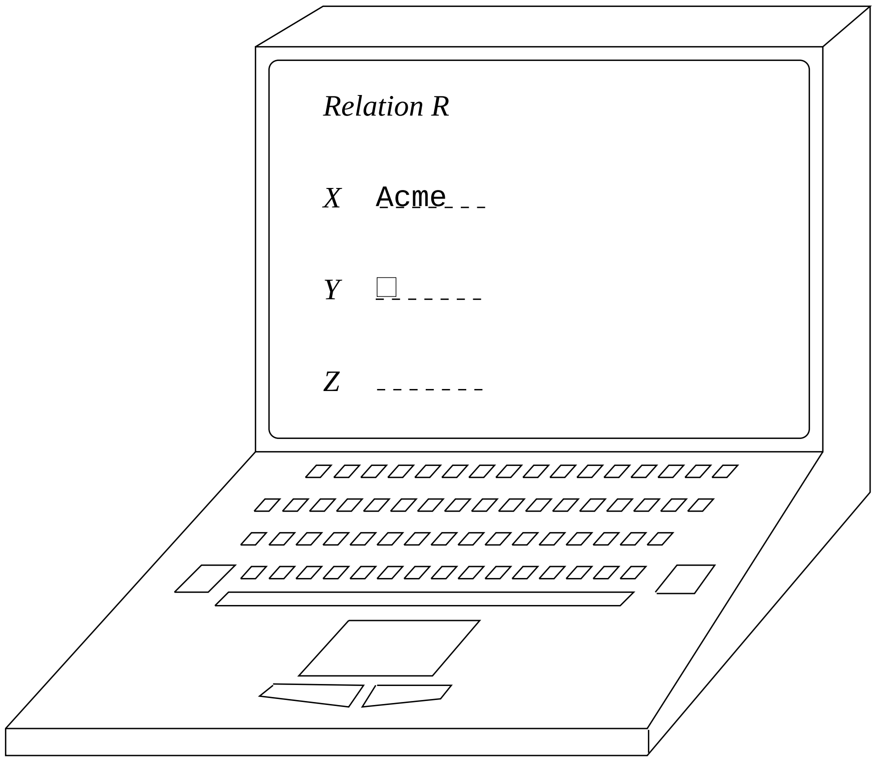


Figure 6: Unary Operators of the Relational Algebra: Edit

editor which presents individual tuples for manipulation and permits addition of new tuples is what hey wants.

Figure 6 shows the **edit** operation from both perspectives. For the programmer user, it looks just like project: the attribute list containing X is an optional indication that the tuples are to be presented to the end user ordered according to X. The **edit** operation does not invoke an algorithm as does project, but opens an edit window for the end user and runs until told to stop.

3 Putting Relations Together—Binary Operations

The binary operations of the relational algebra are extensions of the binary operations on sets: relations are special kinds of sets, and so have operators which are specialized versions of the set operators. Set operators come in two kinds. The first satisfy closure: they produce new sets. They are intersection, union, difference, and symmetric difference of sets. The second kind of set operator are comparisons and produce truth values: equality, subset, disjoint, and their negations.

Relations are closed under both kinds of operator when extended to relations. Figure 7 summarizes the two kinds, called the families of μ-joins and σ-joins, respectively.

- μ-joins (*"set"-valued*)

ijoin $\cap$	**natjoin**
ujoin $\cup$	**ljoin**
sjoin $+$	**rjoin**
djoin $-$	**drjoin**

- σ-joins (*"truth"-valued*)

$\supset$ **gtjoin**	$\supseteq$ **div**	$=$ **eqjoin**	$\subseteq$ **lejoin**	$\subset$ **ltjoin**	$\cap\!\!\!\circ$ **sep**
$\not\supset$ **!gtjoin**	$\not\supseteq$ **!gejoin**	$\neq$ **!eqjoin**	$\not\subseteq$ **!lejoin**	$\not\subset$ **!ltjoin**	$\not\cap\!\!\!\circ$ **icomp**

Figure 7: Binary Operators of the Relational Algebra

Responsibility

(*Agent*	*Item*)
Raman	Micro
Raman	Terminal
Smith	V.C.R.
Hung	Micro

Location

(*Item*	*Floor*)
Micro	1
Terminal	1
Terminal	2
Videodisk	2

Responsibility **ijoin** *Location*

(*Agent*	*Item*	*Floor*)
Raman	Micro	1
Hung	Micro	1
Raman	Terminal	1
Raman	Terminal	2

Figure 8: The Intersection, or Natural, Join

3.1 μ-Joins

We start with the μ-joins. The first and most important member of this family extends set intersection, and is defined, for relations R and S on attribute *sets* W, X, Y, and Z as follows.

- For relations $R(X, Y)$ and $S(Y, Z)$ sharing a common attribute set, Y
  R **ijoin** $S \equiv \{(x, y, z) \mid (x, y) \in R$ and $(y, z) \in S\}$

- For relations $R(W, X)$ and $S(Y, Z)$ sharing no common attribute set
  $R[X$ **ijoin** $Y]S \equiv \{(w, x, y, z) \mid (w, x) \in R$ and $(y, z) \in S$ and $x = y\}$

Figure 8 gives an example.

The *intersection join*, **ijoin**, is also called the natural join, **natjoin**, and is the original binary operator proposed by Codd [Cod70]. It is the fundamental way of combining two relations, and is also, conversely, related to the *decomposition* of a relation into two. Figure 9 shows the graph forms of variants of the two relations, *Responsibility* and *Location*, in such a way as also to be the graph form of their natural join: each edge from *Agent* to a value of *Item* must be considered linked to each edge from the same value of *Item* to *Floor*. The figure also shows the matrix form of the natural join: the `1`s for each value, v, of *Item* form a rectangle of $r_v \times \ell_v$ tuples, where r_v is the number of tuples in *Responsibility* for that value of *Item* and ℓ_v is the number of tuples in *Location*.

The natural join connects together any tuples that share a value of the *join attribute*.

Figure 10 shows how a ternary relation which does not have this rectangular arrangement of tuples cannot be decomposed into two relations that the natural join can put back together as the original relation. If there is no connection between `Micro`, `Hung`, and floor `2` in the ternary relaton, we do not know whether or not to connect `Micro` with floor `2` in *Location*.

The remainder of the μ-joins follow similar lines. The *union join*, **ujoin**, sometimes called the outer join, retains all information in the result, even the tuples that do not match on the join attribute: the unmatched attribute sets below take on *null values*, $\mathcal{DC}$, which we discuss in section 3.5. For *Responsibility* and *Location*, above, figure 11 gives an example to start with.

In general, the union join consists of three disjoint sets of tuples, the *centre*, the *left wing*, and the *right wing*.

- For relations $R(X,Y)$ and $S(Y,Z)$ sharing a common attribute set, Y
  $\textit{centre} \equiv R$ **ijoin** S
  $\textit{left wing} \equiv \{(x,y,\mathcal{DC}) \mid (x,y) \in R \text{ and } \forall z \text{ not } (y,z) \in S\}$
  $\textit{right wing} \equiv \{(\mathcal{DC},y,z) \mid (y,z) \in S \text{ and } \forall x \text{ not } (x,y) \in R\}$

- For relations $R(W,X)$ and $S(Y,Z)$ sharing no common attribute set
  $\textit{centre} \equiv R[X$ **ijoin** $Y]S$
  $\textit{left wing} \equiv \{(w,x,y,\mathcal{DC}) \mid (w,x) \in R \text{ and } x = y \text{ and } \forall z \text{ not } (y,z) \in S\}$
  $\textit{right wing} \equiv \{(\mathcal{DC},x,y,z) \mid (y,z) \in S \text{ and } x = y \text{ and } \forall x \text{ not } (x,y) \in R\}$

Using these, and $\cup$ for set union, we can define, for both cases
R **ujoin** S (or $R[X$ **ujoin** $Y]S$) $\equiv$ *left wing* $\cup$ *centre* $\cup$ *right wing*

An application of the union join could be tallying marks for a course. Suppose the marks for the assignments and the marks for the exam were given as separate relations, each with student as the key. Then the **ijoin** would discard any students who did one but not the other, so they would get no credit at all. The **ujoin** would retain all the data.

The *difference join*, **djoin**, extends set difference.
R **djoin** $S \equiv [X,Y]$ **in** *left wing*
$R[X$ **djoin** $Y]S \equiv [W,X,Y]$ **in** *left wing*
respectively. Note that the attribute set, Z, which has all null values, is not included as an attribute set of the result. Figure 12 shows the example, with *Responsibility* and *Location*,

It is sometimes useful to have a *right difference join*, S **drjoin** $R \equiv R$ **djoin** S.

The *symmetric difference join*, **sjoin**, combines the two difference joins, R **djoin** S and S **djoin** R.
R **sjoin** S (or $R[X$ **sjoin** $Y]S$) $\equiv$ *left wing* $\cup$ *right wing*

Figure 13 shows the example, with *Responsibility* and *Location*,

The symmetric difference join could be applied to the same mark tallying problem as the union join. If a student's name were spelled differently in the two input relations (or there were a similar discrepancy in i.d. number), the **sjoin** would serve as an exception report and could lead to the difference being detected and to correcting the two names to be the same.

Two further joins are sometimes useful although they extend set operations which are trivial. The *left join* extends what would be the set operation that simply returns the left operand. The *right join* corresponds. As the examples in figure 14 show, this operation is not so trivial for relations.

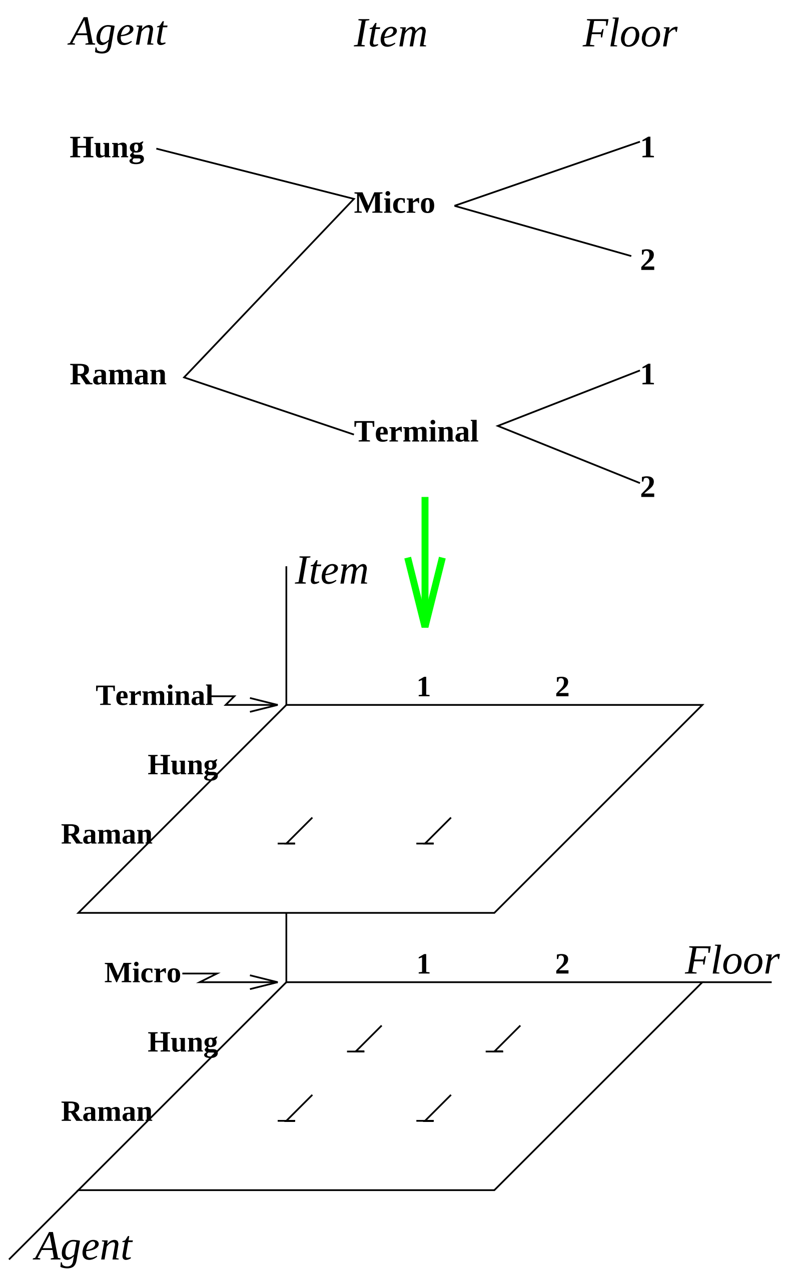


Figure 9: Natural Join and Decomposition

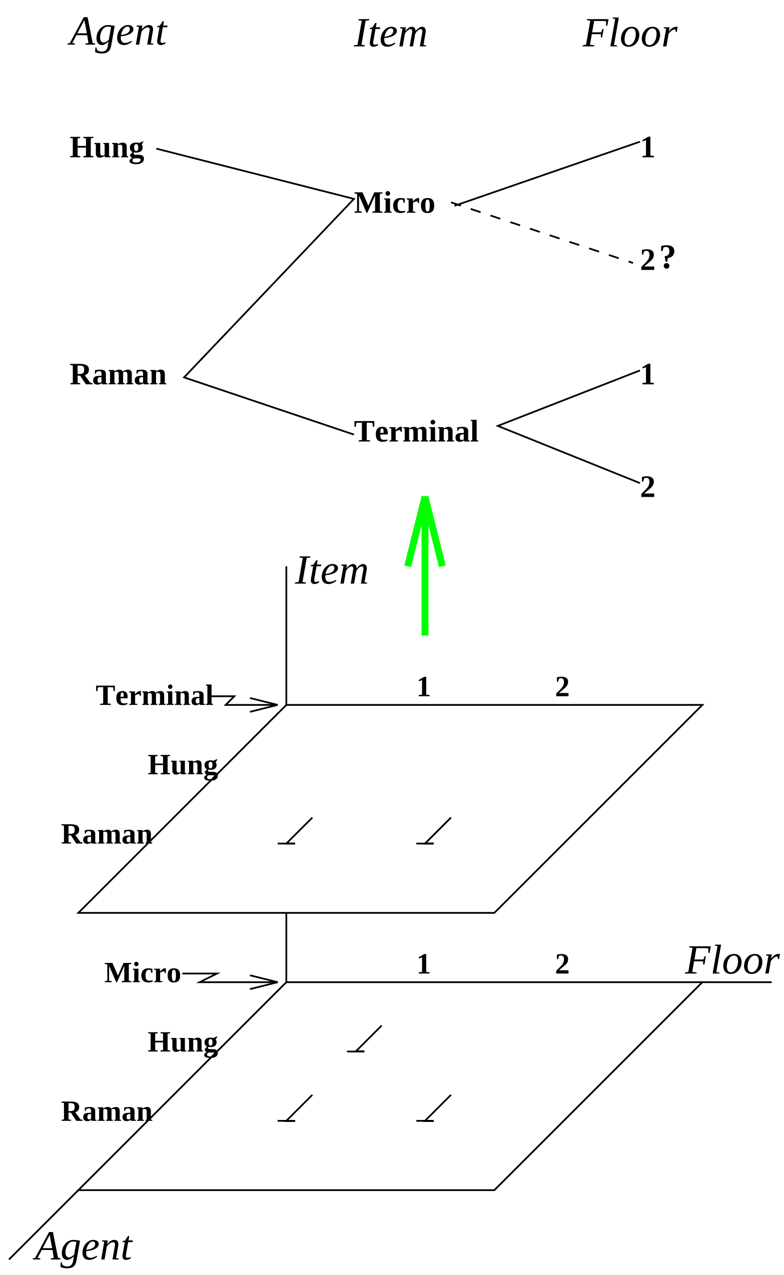


Figure 10: Natural Join and Non-decomposability

Responsibility **ujoin** *Location*

(*Agent*	*Item*	*Floor*)
Raman	Micro	1
Hung	Micro	1
Raman	Terminal	1
Raman	Terminal	2
Raman	Micro	1
Smith	V.C.R.	$\mathcal{DC}$
$\mathcal{DC}$	Videodisk	2

Figure 11: The Union, or Outer, Join

Responsibility **djoin** *Location*

(*Agent*	*Item*	*Floor*)
Smith	V.C.R.	$\mathcal{DC}$

Figure 12: The Difference Join

Responsibility **sjoin** *Location*

(*Agent*	*Item*	*Floor*)
Smith	V.C.R.	$\mathcal{DC}$
$\mathcal{DC}$	Videodisk	2

Figure 13: The Symmetric Difference Join

Responsibility **ljoin** *Location*

(*Agent*	*Item*	*Floor*)
Raman	Micro	1
Hung	Micro	1
Raman	Terminal	1
Raman	Terminal	2
Smith	V.C.R.	$\mathcal{DC}$

Responsibility **rjoin** *Location*

(*Agent*	*Item*	*Floor*)
Raman	Micro	1
Hung	Micro	1
Raman	Terminal	1
Raman	Terminal	2
$\mathcal{DC}$	Videodisk	2

Figure 14: The Left and Right Joins

Here are the definitions.

R **ljoin** S (or $R[X$ **ljoin** $Y]S$) $\equiv$ *left wing* $\cup$ *centre*

R **rjoin** S (or $R[X$ **rjoin** $Y]S$) $\equiv$ *centre* $\cup$ *right wing*

In summary, except for **djoin** (and **drjoin**), the μ-joins all result in a relation whose attributes are the union of the attributes of the two operands. Here are examples of the notation used in Aldat for μ-joins.

(join $\equiv$ **ijoin**, **ujoin**, **sjoin**, **ljoin**, **rjoin**)

1. Assigning the result

   Warehouse <− *Responsibility* **ijoin** *Location*;

2. Join on common attributes

$$R(X, Y),\ S(Y, Z) \quad T <\!- R \,\text{join}\, S; \quad T(X, Y, Z)$$

$$T <\!- R \ \textbf{djoin}\ S; \quad T(X, Y)$$

3. Join on different attributes

$$Q(W, X),\ S(Y, Z) \quad T <\!- Q[X \,\text{join}\, Y]S; \quad T(W, X, Y, Z)$$

   (X and Y are aliases in T.)

   N.B. Q **ijoin** S gives Cartesian product: X, Y are *not* aliases.

4. Join on several attributes

$$U(A, B, C, X, Y, Z),\ V(X, Y, Z, D, E)$$

$$T <\!- U \,\text{join}\, V;$$

$$T(A, B, C, X, Y, Z, D, E)$$

3.2 σ-Joins

The σ-joins extend the truth-valued comparison operations on sets to relations by applying them to each set of values of the join attribute for each of the other values in the two relations. We can show this for the same relations, *Responsibility* and *Location*, that illustrated the μ-joins. Figure 15 shows the operands and the result for the *superset join*, written **sup**, **div**, or $\supseteq$.

All relations are shown in matrix form, with three two-dimensional matrices making up the "back", "side", and "floor" of a box. The "back" is the four tuples of *Responsibility*, the "side" is the four tuples of *Location*, and the "floor" is the result, which we see has only one tuple. This tuple is arrived at by comparing each of the three sets of *Item*s (the common, or join, attribute) in *Responsibility* with each of the two sets of *Item*s in *Location*: only the *Item*s that `Raman` is responsible for form a superset of the *Item*s on *Floor* `1`.

This join may be interpreted as answering the query, "find *Agent*s and *Floorp* such that the *Agent* is responsible for all *Item*s on the *Floor*". This universal quantifier, *all*, is what led Codd [Cod72] to invent the *division* operator for the relational algebra: it was needed to make the algebra equivalent to a limited form of the first-order predicate calculus, which supports universal and existential quantification. Codd's division is a strictly special case of the superset join, in which the right-hand operand has no other attributes than the join attributes.

We can define the σ-joins using the following notation. In relations $R(W, X)$ and $S(Y, Z)$, R_w is the set of values of X associated by R with a given value, w, of W, and S_z is the set of

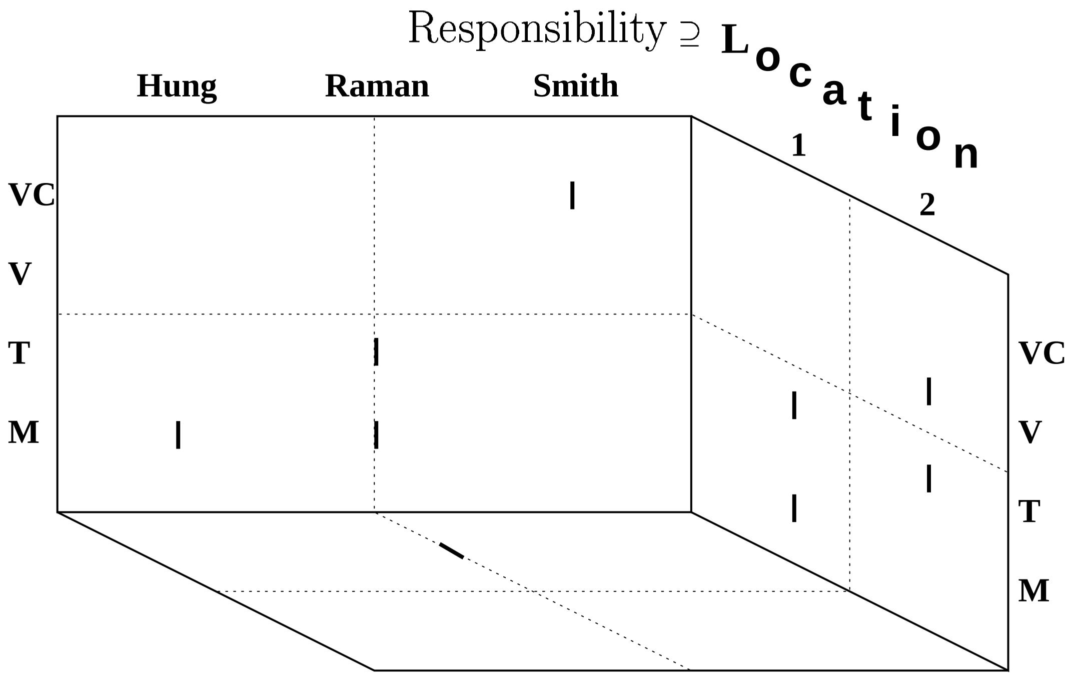


Figure 15: The Superset Join, or Division

values of Y associated by S with a given value, z, of Z. If W and X are disjoint sets of the attributes of R, and Y and Z are disjoint sets of the attributes of S, the following definitions are general, and even allow for X and Y to be the same set of attributes. X and Y must be at least compatible attribute sets.

R **sup** $S \equiv \{(w, z) \mid R_w \supseteq S_z\}$

The second important σ-join is **icomp**, which extends the notion of set overlap.

R **icomp** $S \equiv \{(w, z) \mid R_w \not\!\phi\!\!\backslash S_z\}$

where $R \not\!\phi\!\!\backslash S$ means $R \cap S \neq \phi$ and introduces a comparison operator meaning that the two operand sets overlap. Figure 16 illustrates this.

The name *natural composition* is from Codd [Cod70], who proposed this join to convery existential quantification. The query answered by the example is "find *Agent*s and *Floor*s such that the *Agent* is responsible for some of the *Item*s on the *Floor*" (or, symmetrically, "such that the *Floor* holds some of the *Item*s looked after by the *Agent*").

The operations in the μ-join and σ-join families are not independent of each other, either within their families or across families. An important link across families is between natural join and natural composition. Figure 17 shows that projecting the result of the natural join on the non-join attributes gives the result of the natural composition of the two operand relations. The projection can be likened to shining a light down through the planes of the matrix form of the join to give shadows of the 1s.

Natural composition is the obvious operator to use when the join attribute is not needed in the result, and especially when the natural join is impossible because of attribute ambiguities. For example, suppose we want to find *Grandparent* given *Parent*.

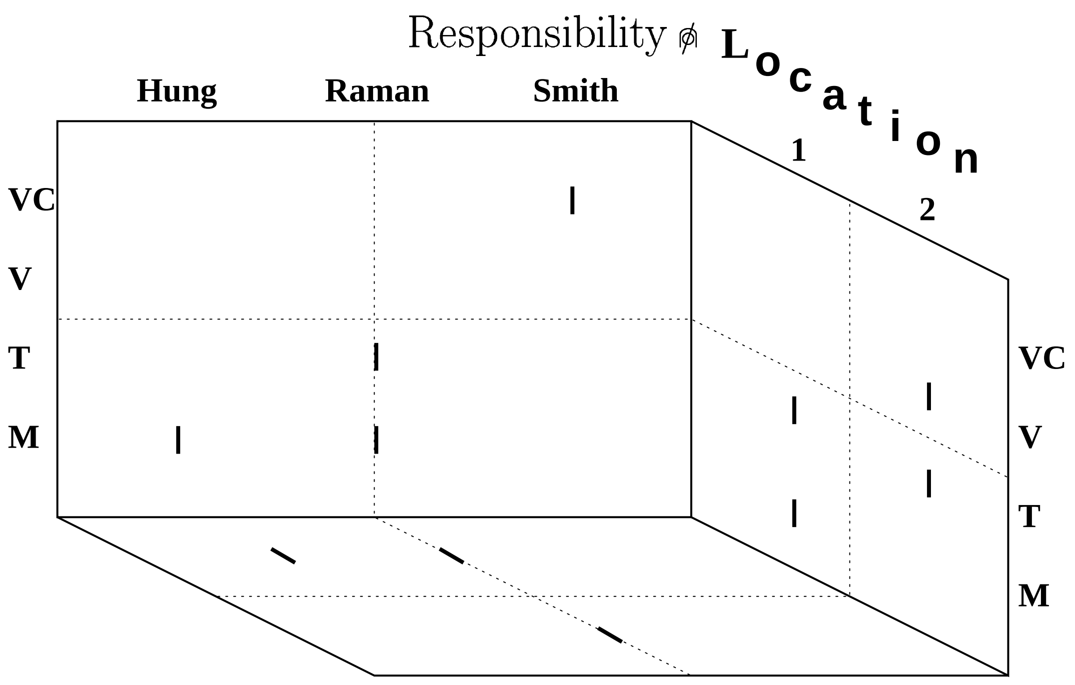


Figure 16: The Overlap Join, or Natural Composition

Parent(*Sr*	*Jr*)
Sam	Pete
Pete	Sue
Pete	Joe

Grandparent(*Sr*	*Jr*)
Sam	Sue
Sam	Joe

Grandparent <– *Parent*[*Jr* **icomp** *Sr*]*Parent*;

The set overlap symbol used in the natural composition has a bar through it: it is the complement of disjointness, **sep**. (Each of the twelve σ-joins has a complement, one of the others.)

$$R \textbf{ sep } S \equiv \{(w,z) \mid R_w \Cap S_z\}$$

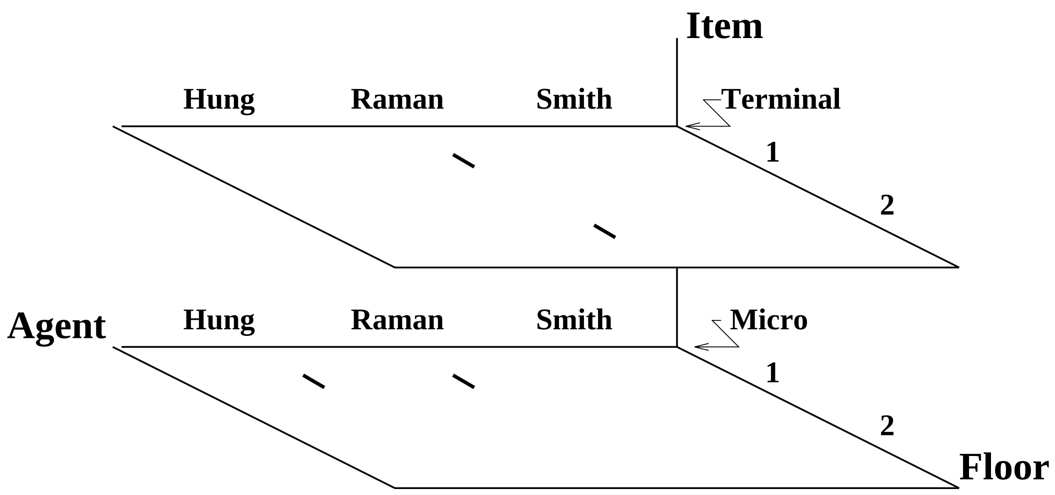
Figure 18 shows that the tuples in the result complement the tuples for **icomp**.

The remaining nine σ-joins correspond to the set comparisons $\supset, =, \subseteq, \subset$, and complements $\not\supset$, $\not\supseteq$, $\neq$, $\not\subseteq$, and $\not\subset$, below and as shown in figure 7.

$$\begin{aligned}
R \textbf{ gtjoin } S &\equiv \{(w,z) \mid R_w \supset S_z\}\\
R \textbf{ eqjoin } S &\equiv \{(w,z) \mid R_w = S_z\}\\
R \textbf{ lejoin } S &\equiv \{(w,z) \mid R_w \subseteq S_z\}\\
R \textbf{ ltjoin } S &\equiv \{(w,z) \mid R_w \subset S_z\}\\
R \textbf{ !gtjoin } S &\equiv \{(w,z) \mid R_w \not\supset S_z\}\\
R \textbf{ !gejoin } S &\equiv \{(w,z) \mid R_w \not\supseteq S_z\}\\
R \textbf{ !eqjoin } S &\equiv \{(w,z) \mid R_w \neq S_z\}\\
R \textbf{ !lejoin } S &\equiv \{(w,z) \mid R_w \not\subseteq S_z\}\\
R \textbf{ !ltjoin } S &\equiv \{(w,z) \mid R_w \not\subset S_z\}
\end{aligned}$$

Syntactically, the attribute set resulting from any σ-join is the symmetric difference of the input attribute sets. The notation parallels that for μ-joins, above.

Responsibility **ijoin** *Location*



Responsibility **icomp** *Location*
[*Agent, Floor*] **in** (*Responsibility* **ijoin** *Location*)

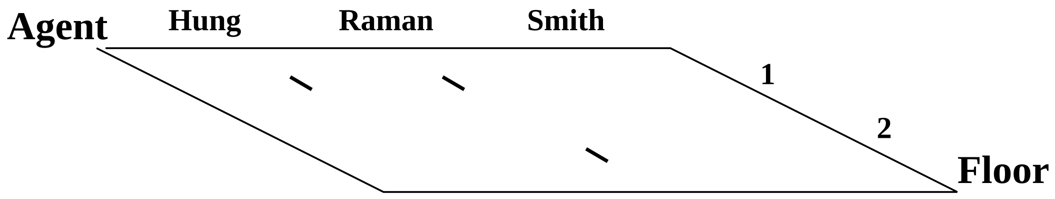


Figure 17: Natural Join and Natural Composition

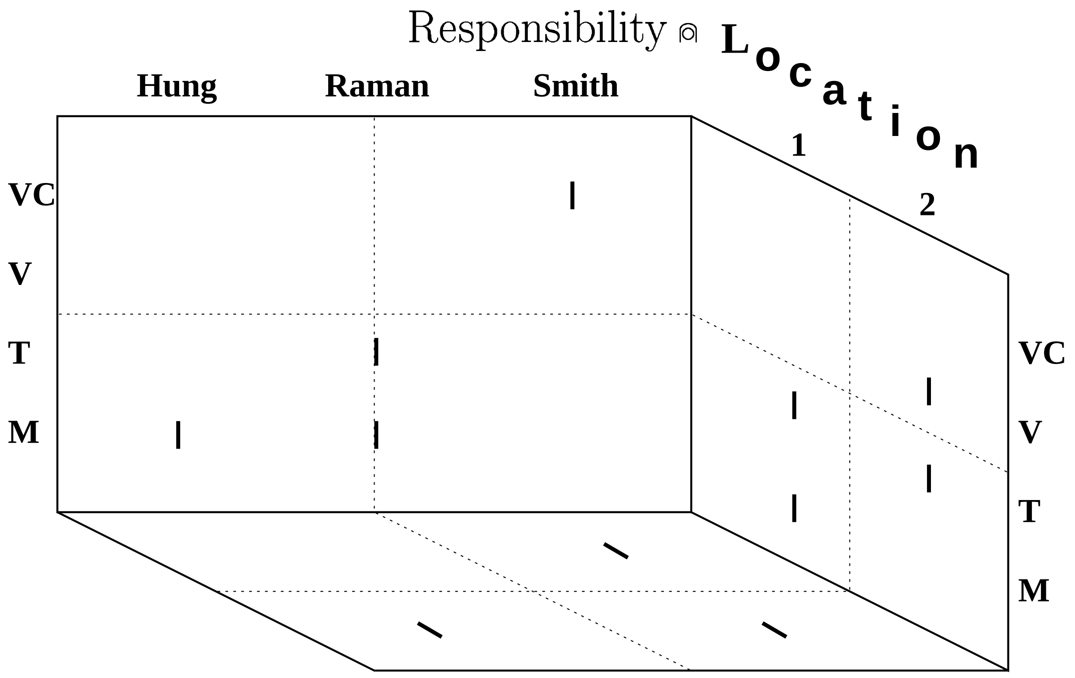


Figure 18: Disjointness

Here are examples (join = [**!**]**gtjoin**, **div**, **!gejoin**, [**!**]**eqjoin**, [**!**]**lejoin**, [**!**]**ltjoin**, **sep**, **icomp**).

1. Join on common attributes

$$R(X,Y),\ S(Y,Z) \quad T <\!\!- R\,\text{join}\,S; \quad T(X,Z)$$

2. Join on different attributes

$$Q(W,X),\ S(Y,Z) \quad T <\!\!- Q[X\ \text{join}\,Y]S; \quad T(W,Z)$$

3. Join on several attributes

$$U(A,B,C,X,Y,Z),\ V(X,Y,Z,D,E)$$
$$T <\!\!- U\,\text{join}\,V;$$
$$T(A,B,C,D,E)$$

As a consequence, we must deal with the special case that both operands have the same attribute sets. This is the special case of set comparison.

$$R(Y),\ S(Y) \quad T <\!\!- R\,\text{join}\,S; \quad T()$$

The result relation is *nullary*: it has no attributes. This must be interpreted as Boolean, because the special case of set comparison is truth-valued. Fortunately, this is an easy abstraction to make. A nullary relation can have only two values: either it is empty or it is not empty. So Booleans are relations, and closure is saved for the σ-joins.

An important practical consideration when performing a σ-join is to remember to project the operand relations on the relevant attributes only. For instance, referring to section 2 of chapter 1.1, if we had a relation

NYPenn(	*Cust*	)
	`Pennsylvania Railroad`	
	`New York Central`	

and we wished to find out which *Sales*people had processed orders from both of those railways,

NYPennPeople <− ([*Cust, Sales*] **in** *Orders*) $\supseteq$ *NYPenn*;

would give the right answer,

NYPennPeople(	*Sales*	)
	`Hannah Trainman`	

whereas

WrongNYPennPeople <− *Orders* $\supseteq$ *NYPenn*;

would not:

WrongNYPennPeople(	*Ord#*	*Sales*	)

is empty.

3.3 Relational Expressions

The individual operators of the relational algebra support simple *queries*, such as "what agents are responsible for micros?" (see figure 5 on page 4). Combining operators permits more elaborate queries. But the relational algebra is a basis for high-level *programming*, rather than for end-user querying, so we must look at at least one simple combination from this perspective. [1]

Consider the query "what floors contain the items Raman is responsible for?". This requires us to put together two relations, in order to link *Agent* with *Floor*, which we will select and project on, respectively (see figure 8 on page 6). The need to put *Responsibility* and *Location* together should lead us to think first of the natural join, and reflection should confirm that this is the correct join. Thus, an expression of the query could be

[*Floor*] **where** *Agent*=`Raman` **in** (*Responsibility* **ijoin** *Location*)

where the parentheses show that the natural join is performed first, followed by the t-selector. (A convention of operator precedence might eliminate the need for parentheses.)

This illustrates the closure property of the relational algebra. Expressions may be built up of independent operations. This is an aspect of programming. The same result may be obtained by different expressions. Different expressions may execute at different speeds. *Optimization* is the process of finding the fastest expression for any given result. We will discuss the simple optimization a programmer may achieve just by being careful, as hey would avoid putting an operation inside a loop if it can be done outside.

If we join *Responsibility* and *Location* before doing the selection, we may have a quite large intermediate result: the natural join could have many more tuples than the two operands, and could in the worst case be their Cartesian product. Doing the selection first always reduces the number of tuples, and this should be a programming reflex.

[*Floor*] **in** (*Location* **ijoin where** *Agent*=`Raman` **in** *Responsibility*)

Now the join will involve only the two tuples from *Responsibility* that pertain to `Raman`, and will have only two tuples instead of four.

[1] End-user needs are met by the editors that can be provided, and their user interfaces.

Notice that we did not also move the projection inside the join. We could have, but this does not guarantee faster exection the way moving the selection inside always does: projection requires $\mathcal{O}(n \log n)$ time, for n blocks of data, because it must sort to eliminate duplicates. To determine whether or not it is faster to do the projection before the join would require an analysis of the data structures and algorithms underlying the operations, and would depend on the data in the relations.

However, since the above solution requires three operations (select, then join, then project), it could be made more elegant for the programmer, and this is an important concern in the absence of sure knowledge of execution speeds.

Location **icomp** [*Floor, Item*] **where** *Agent*=`Raman` **in** *Responsibility*

(We have replaced the natural join by natural composition to save a final projection on *Floor*.)

(Those used to certain commercial query languages might be tempted to say

[*Floor*] **where** *Item* $\in$ ([*Item*] **where** *Agent*=`Raman` **in** *Responsibility*) **in** *Location*

replacing the join by a set membership test, $\in$. Such a expression is not a part of first-normal-form relational algebra. Not only does it mix levels between attribute and relation, but also it leads to expectations that one may write, say, for relation $R(A, B)$

where $A \in B$ **in** R

in which B must be a set of values (of the same type as A). This would violate the restriction to first normal form. Such notation will reappear legitimately when we consider nested relations.)

3.4 Relational Recursion

Problems involving repeated applications of relational algebra operators could be solved by providing a looping construct in the language. But we can go a long way without new syntax if we allow recursive definitions of relations.

We start with an example. An *ancestor* **is** a *parent* **or** the *ancestor* **of** an *parent*. This is a compact way of saying an *ancestor* is a *parent* or a *parent* of a *parent* (grandparent) or a *parent* of a *parent* of a *parent* (great-grandparent) or ... It is a recursive definition.

If we are given

relation *Parent*(*Sr, Jr*) <− ... ;
relation *Ancestor*(*Sr, Jr*);

in which *Parent*(*Sr, Jr*) already holds data such as

Parent(*Sr*	*Jr*)
Joe	Sue
May	Sam
Sue	Max
Sam	Max
Max	Ted
Max	Ann
Ted	Jim
Win	Jim
Ted	Nan

and the attributes of *Ancestor* have been declared, then it is easy to specify the contents of *Ancestor* as a recursive view.

[New]Facts	*Horn*			
(Concl	*Rule#*	*Ante*	*Concl*	)
`lays eggs`	1	`lays eggs`	`is bird`	
`has feathers`	1	`has feathers`	`is bird`	
`swims`	2	`flies`	`is bird`	
`is brown`	2	`is not mammal`	`is bird`	
——	3	`is bird`	`is duck`	
`is bird`	3	`swims`	`is duck`	
——	3	`is brown`	`is duck`	
`is duck`	4	`is bird`	`is duck`	
	4	`swims`	`is duck`	
	4	`is green`	`is duck`	
	4	`is red`	`is duck`	
	5	`is duck`	`migrates`	
	5	`is not tame`	`migrates`	

NewFacts **is** *Facts* **ujoin** [*Concl*] **in** (*NewFacts*[*Concl* $\supseteq$ *Ante*]*Horn*)

Figure 19: Horn Clauses: An Inference Engine

Ancestor **is** *Parent* **ujoin**
Parent[*Jr* **icomp** *Sr*]*Ancestor*;

As we said in Section 1, nothing is evaluated when this view is defined. When an assignment is done from *Ancestor*, or it is to be printed, the following loop is executed.

```
Ancestor <−  $\phi$                       // empty
repeat  Test <− Ancestor
    Ancestor <− Parent ujoin
        Ancestor[Jr icomp Sr]Parent
until Ancestor=Test
```

which puts *Parent* into *Ancestor* the first time around, adds grandparent the second time, great-grandparent the third, and so on until the **icomp** *Ancestor* with *Parent* adds no new tuples and the iteration halts.

(This implementation, which is the actual execution used by Aldat, also repeatedly adds parent, grandparent, and all the intermediate generations, to *Ancestor*: it is horribly inefficient. It is even worse than what in the literature is termed the "naive implementation" [BR86]. But it turns out [CLM89] that the recursive view can be rewritten so as to reproduce the most effective algorithms in the literature, even with this subnaive underlying implementation.)

An Inference Engine

The σ-join is very useful in other relational recursions. Here is a very simple inference engine, using *Horn clauses*. These are logical implications in which a conjunction (**and**) of *antecedents* imply a single *conclusion*. For a conclusion to hold, *all* the antecedents must hold, so it seems that a **sup**erset join will be involved. Further, disjunction (**or**) is expressed by writing two Horn cluases with the same conclusion. In figure 19, *Horn*(*Rule#, Ante, Concl*) contains five examples. The *Rule#* is necessary to distinguish two or more Horn clauses with the same conclusion; each Horn clause requires several tuples, one for each antecedent.

The *Facts* relation is the first three tuples shown, and gives the starting point for the inference. The *NewFacts* relation is *Facts* after the first itereation, then adds the next two tuples in two iterations after a rule "fires" on each iteration. So, in this example, we start with knowing that (it) `lays eggs` and `has feathers` and `swims`. The first two tell us, through rule `3`, that it `is` a `bird`. This, with `swims`, tells us, through rule `4`, that it `is` a `duck`.

Notice that *Horn* is a second example of a relation in which individual tuples do not have much meaning, but must be grouped to be meaningful. The grouping facility of the σ-join is essential to support this kind of interpretation of a relation. If there were no operations on groups of tuples, the meaning of such groups would be impotent.

(With a little work (one person-month) this one-line inference engine can be expanded to sophisticated one, together with a full "expert system shell", *Relixpert* [Mer91], which requires only 200 lines of Aldat, a database language developed with no consideration of artificial intelligence applications.)

More Logic

If we go back further in time, we find the *syllogisms* of Aristotle and the medieval churchmen. A syllogism has two premises from which it draws a conclusion. An example is

All philosophers are human
All humans are mortal

All philosophers are mortal

This example can be mapped onto the transitivity of set containment.

$$\begin{array}{c} P \subseteq H \\ H \subseteq M \\ \hline P \subseteq M \end{array}$$

Other examples of syllogisms involve the existential quantifier.

All philosophers are clever
All philosophers are human

Some humans are clever

All clever beings play chess
Some humans are clever

Some humans play chess

Such premises and conclusions can be mapped onto set overlap.

$$H \mathrel{\cap\!\!\!\!\!\not\circ} C$$

$$H \mathrel{\cap\!\!\!\!\!\not\circ} PC$$

Figure 20 shows the full set of rules that govern these two relationships, from which appropriate closure operations can be used to infer any syllogism or collection of syllogisms. We leave it as an exercise for the reader to formulate the relationships and the syllogistic inference engine using the relational algebra.

A, E	Universally quantified			
	A	every X is Y	$X \subseteq Y$	$Y' \subseteq X'$
	E	no X is Y	$Y \subseteq X'$	$X \subseteq Y'$
I, O	Existentially quantified			
	I	some X is Y	$X \cap\!\!\!\!\!\!\not\emptyset\, Y$	$Y \cap\!\!\!\!\!\!\not\emptyset\, X$
	O	some X is not Y	$X \cap\!\!\!\!\!\!\not\emptyset\, Y'$	$Y' \cap\!\!\!\!\!\!\not\emptyset\, X$

- **A, E** call $\subseteq$ *isA*
  rules:
  - antisymmetric, transitive: $\therefore$ closure
- **I, O** combine with **A, E**: call $\cap\!\!\!\!\!\!\not\emptyset$ *laps*
  rules:
  - symmetric
  - X *laps* Y & Y *isA* $Z \Rightarrow X$ *laps* Z
    (*laps* is closed under **icomp** with *isA*)
  - X *isA* Y & X *isA* $Z \Rightarrow Y$ *laps* Z
  - X *isA* $Y \Rightarrow X$ *laps* Y

Figure 20: Syllogisms

(The churchmen used the four Latin vowels as mnemonic aids, representing each possible syllogism by a word containing three vowels. For instance, the above three examples are, respectively, *Barbara, Darapti,* and *Darii.* There are a dozen or so more, with many formulations about what combinations are legitimate. Your Aldat syllogism code should replace all that. [KK62] pp.67–81)

Dodgson [Car96] provides many delightful examples of a special case of the above, restricted to the *isA* hierarchy. Figure 21 gives one, with the *isA* formulation and the final conclusion (which is one of many). As with all of Dodgson's examples, this forms a chain, and the interesting conclusion is the one that links the beginning and the end of the chain. Again, we leave to the reader the construction of the inference engine using the relational algebra.

3.5 Null Values

The μ-joins required the introduction of the $\mathcal{DC}$ null value. This has the significancs "don't care". A second type of null value, $\mathcal{DK}$, means "don't know". The first describes irrelevance, the second missing data.

The treatment of null values in this section is approximate. The two null values themselves are simplifications: an early study [B*et al.*62] reported fourteen different types of null value. We will see that some logical inconsistences arise, but we propose to live with them in the absence of conclusive simple results on nulls in general.

The "don't care" null value is taken to behave as a special value whose main property is that it should have no effect on scalar operations applied to the values of the attributes.

- $\mathcal{DC}$ is the right and left identity for any binary operator that preserves type:
- $x + \mathcal{DC} = x$, $\mathcal{DC} + x = x$, $x \times \mathcal{DC} = x$, ...

	Isa (*Subj*	*Obj*)
1. No shark ever doubts that it is well fitted out.	S	F
2. A fish, that cannot dance a minuet, is contemptible.	M´	C
3. No fish is quite certain that it is well fited out, unless it has thre rows of teeth.	F	T
4. All fishes, except sharks, are kind to children.	S´	K
5. No heavy fish can dance a minuet.	H	M´
6. A fish with three rows of teeth is not to be despised.	T	C´
∴ No heavy fish is unkind to children.	H	K

Figure 21: *isA* Syllogisms

- $\mathcal{DC} - x = \mathcal{DC} + (-x) = -x$ (by caveat: practice could change this)
- $\mathcal{DC} \div x = \mathcal{DC} \times (\div x) = \div x$
- Unary operations on $\mathcal{DC}$ are ignored: $-\mathcal{DC} = \mathcal{DC}, \neg\mathcal{DC} = \mathcal{DC}$
- $x \ \kappa \ \mathcal{DC}$ is $\mathcal{DC}$ for any comparison, $\quad \kappa \in \{<, \leq, =, \geq, >, \neq\}$

There are number of consequences of this design which we must watch out for.

- Note that $a\mathcal{DC} + by = a + by$, not by; and $ay = a(\mathcal{DC} + y) = a\mathcal{DC} + ay = a + ay$
- Because $x = \mathcal{DC} = \mathcal{DC}$, the natural join is not well defined if a join attribute is $\mathcal{DC}$: we get Codd's "maybe" join [Cod75].

The stipulation that $x \ \kappa \ \mathcal{DC}$ is $\mathcal{DC}$, and the consequent maybe join is unfortunate, but

- if we made $x \ \kappa \ \mathcal{DC}$ **false**:
  true $= \neg(x < \mathcal{DC}) = (x \geq \mathcal{DC}) =$ **false**;
- if we made $x \ \kappa \ \mathcal{DC}$ $\mathcal{DC}$ for $\kappa \in \{<, \leq, \geq, >\}$,
  $x = \mathcal{DC}$ **false**, and $x \neq \mathcal{DC}$ **true**, (ordinary x):
  $$\begin{aligned}\mathcal{DC} = (x < \mathcal{DC}) &= (x \leq \mathcal{DC}) \wedge \neg(x = \mathcal{DC}) \\ &= ((x < \mathcal{DC}) \vee (x = \mathcal{DC})) \wedge \neg(x = \mathcal{DC}) \\ &= (\mathcal{DC} \vee \textbf{false}) \wedge \textbf{true} \\ &= \textbf{false}.\end{aligned}$$

The "don't know" null value, $\mathcal{DK}$, is less straightforward. It is not a special value, but is best thought of as a *variable* with a range of all the non-null values of the domain.

- $x \ \kappa \ \mathcal{DK} = \mathcal{DK}$, $\kappa \in \{<, \leq, =, \geq, >, \neq\}$ Again, we get the "maybe" join.
- Operators with $\mathcal{DK}$ almost always give $\mathcal{DK}$:
  $\neg\mathcal{DK} = \mathcal{DK}$, $\mathcal{DK} - x = \mathcal{DK}$, $x \times \mathcal{DK} = \mathcal{DK}$, x **max** $\mathcal{DK} = \mathcal{DK}$, etc.

κ	x	$\mathcal{DC}$	$\mathcal{DK}$	$\kappa \in \{<, \leq, =, \geq, >, \neq\}$
x		$\mathcal{DC}$	$\mathcal{DK}$	
$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$	
$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DC}$	$\mathcal{DK}$	

and	F	T	$\mathcal{DC}$	$\mathcal{DK}$	**or**	F	T	$\mathcal{DC}$	$\mathcal{DK}$	**not**	
F	F	F	F	F	F	F	T	F	$\mathcal{DK}$	F	T
T	F	T	T	$\mathcal{DK}$	T	T	T	T	T	T	F
$\mathcal{DC}$	F	T	$\mathcal{DC}$	$\mathcal{DK}$	$\mathcal{DC}$	F	T	$\mathcal{DC}$	$\mathcal{DK}$	$\mathcal{DC}$	$\mathcal{DC}$
$\mathcal{DK}$	F	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	T	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$

$+$	x	$\mathcal{DC}$	$\mathcal{DK}$	$-$	x	$\mathcal{DC}$	$\mathcal{DK}$	$-$	
x		x	$\mathcal{DK}$	x		x	$\mathcal{DK}$	x	$-x$
$\mathcal{DC}$	x	$\mathcal{DC}$	$\mathcal{DK}$	$\mathcal{DC}$	$-x$	$\mathcal{DC}$	$\mathcal{DK}$	$\mathcal{DC}$	$\mathcal{DC}$
$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$

$\subset$	$\{x\}$	$\{x, \mathcal{DC}\}$	$\{x, \mathcal{DK}\}$	$\{x, \mathcal{DC}, \mathcal{DK}\}$
$\{x\}$	F	T	$\mathcal{DK}$	T
$\{x, \mathcal{DC}\}$	F	F	F	$\mathcal{DK}$
$\{x, \mathcal{DK}\}$	F	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$
$\{x, \mathcal{DC}, \mathcal{DK}\}$	F	F	F	$\mathcal{DK}$

$\subseteq$	$\{x\}$	$\{x, \mathcal{DC}\}$	$\{x, \mathcal{DK}\}$	$\{x, \mathcal{DC}, \mathcal{DK}\}$
$\{x\}$	T	T	T	T
$\{x, \mathcal{DC}\}$	F	$\mathcal{DC}$	F	$\mathcal{DC}$
$\{x, \mathcal{DK}\}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$	$\mathcal{DK}$
$\{x, \mathcal{DC}, \mathcal{DK}\}$	F	$\mathcal{DK}$	F	$\mathcal{DK}$

Figure 22: Null Values as Four-Valued Logic

- **true or** $\mathcal{DK}$ = **true**,
  false and $\mathcal{DK}$ = **false**

Furthermore, each $\mathcal{DK}$ ranges independently.

- $\mathcal{DK} \vee \neg\mathcal{DK} = \mathcal{DK}$, and is not a tautology.
- So we cannot know that two people have the same age, but not know the age.

A full treatment might use a *set of variables* for $\mathcal{DK}$, and take into account *partial knowledge*, e.g., "his age is between 25 and 40".

This is summarized by figure 22, which pretends that $\mathcal{DC}$ and $\mathcal{DK}$ *are* extra values which augment each language type.

References

[B*et al.*62] R. Bosak and *et al.* An information algebra. phase I report—language structure group of the CODASYL development committee. *Comm. ACM*, 5(4):190–204, April 1962.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Internat. Conf. on the Management of Data*, pages 16–52, May 1986.

[Car96] Lewis Carroll. *Symbolic Logic.* [s.n.], London, 1896.

[CLM89] 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.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, June 1970.

[Cod72] 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.

[Cod75] E. F. Codd. Understanding relations (installment #7). *ACM FDT*, 7(3–4):23–8, 1975. (Subsequently ACM SIGMOD Record).

[KK62] William Kneale and Martha Kneale. *The Development of Logic.* Oxford University Press, London, 1962.

[Mer91] 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.