

On the Design of an Indexical Query Language

Joey Paquet, John Plaise

Département d'informatique, Université Laval

Ste-Foy, Québec, Canada

`{jpaquet,plaise}@iad.ift.ulaval.ca`

Abstract

For the past twenty years, the relational model has been the norm for defining databases. The model is simple to understand, has a clear semantics and is easy to use.

However, relational algebra is unable to solve certain problems. In particular, it is impossible to express such simple concepts as the transitive closure of a binary relation. Furthermore, the relational algebra itself offers no method for dealing with time in its various forms.

Instances of relations can be seen as two-dimensional arrays with no explicit bounds, i.e. two-dimensional streams. Therefore a relation in a given database can be coded in Lucid as a pair consisting of a one-dimensional stream defining the relation scheme and a two-dimensional stream defining the instance.

Once these concepts have been proposed, we show that certain problems that have been difficult to solve within the standard relational model, such as deductive and historic databases, can be easily solved in the dataflow model, without putting into question the basic notion of a relation.

1 Introduction

Databases form an increasingly widespread use of computer technology in our society. They are used not just for storing data, but also for retrieving it. The relational model of data and the relational algebra are the most commonly used forms of data storage and querying. However, the relational algebra has proven to be ineffective in the resolution of certain queries, namely the transitive closure of a binary relation [1].

To solve this problem, we propose to combine the expressive power of relational algebra and the iterative powers of Lucid, using the following analogy: a relation is a bidimensional finite array and a dataflow is a multidimensional infinite array. Viewed in this manner, a relation is a special case of a dataflow and, as we shall see, the Lucid dataflow programming language is well suited for the implementation of a query language.

A database relation can be represented as two dataflows: the *instance dataflow* (Figure 1) represents the relation instance and the *scheme dataflow* (Figure 2) represents the database relation scheme. We call the whole a *relation couple*. Therefore, for each relation instance r in the database there is an associated relation couple, written $\langle R, r \rangle$.

Supposing this interpretation of a database as relation couples, means of translating relational algebra expressions into Lucid expressions that will compute these relation couples must be provided. In the next section, we present the syntax and semantics of the relational algebra. The following section, after a brief recall of the basic Lucid operators, presents the translation from relational algebra into Lucid. A section giving examples is followed by a presentation of the transitive closure of a relation in Lucid. We conclude with a discussion on temporal and spatial databases.

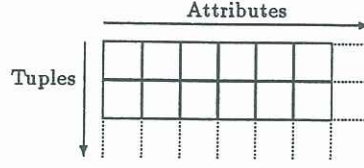


Figure 1: Instance Dataflow

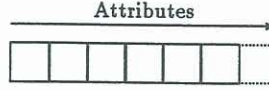


Figure 2: Scheme Dataflow

2 Relational algebra: syntax and semantics

The relational algebra is a query language that processes databases formulated according to the relational model of data. In this model, an occurrence r of a relation consists of a set of tuples r defined over a set of attributes R , where R is the *relation scheme* and r is the *relation instance*.

We suppose that the readers are familiar with the relational algebra, so the syntax and semantics are presented without any explanation. See, for example, [1] for a full discussion.

2.1 Syntax

The syntax given below defines relation occurrences r , functional dependencies FD , boolean formulas F and sequences of attributes X . The terminals include constants c and attribute names A ; θ stands for any comparison operator; f stands for a substitution function among attributes and FD stands for the set of functional dependencies defined on the database.

$r ::= <R, r>$	$FD ::= (X \rightarrow X)^*$
$r \cup r$	$f ::= X \leftarrow X$
$r \cap$	$F ::= A \theta c$
$r - r$	$A \theta A$
$\sigma_F(r)$	$\neg F$
$\pi_X(r)$	$F \vee F$
$\rho_f(r)$	$F \vee F$
$r \bowtie r$	$X ::= A^*$
$\text{chase}_{FD}(r)$	

2.2 Semantics

To simplify the presentation below, we suppose that $r \equiv \langle R, r \rangle$.

$r \cup s$	$= \{t_i \mid (t_i \in r \vee t_i \in s)\}$	union
$r \cap s$	$= \{t_i \mid (t_i \in r \wedge t_i \in s)\}$	intersection
$r - s$	$= \{t_i \mid (t_i \in r \wedge t_i \notin s)\}$	difference
$\sigma_F(r)$	$= \{t \in r \mid F(t) = \text{true}\}$	selection
$\pi_X(r)$	$= \{t[X \cap R] \mid t \in r\}$	projection
$\rho_f(r)$	$= \{t \mid (\exists t' \in r) t'[A] = t[f(A)], \forall A \in R\}$	renaming
$r \bowtie s$	$= \{t[YZ] \mid (\exists t_0 \in r, t_1 \in s) t[XY] = t_0[XY] \wedge t[XZ] = t_1[XZ]\}$	natural join
$\Omega_{FD}(r)$	$= \{t' \mid (\exists t \in r) t' \text{ is the result of enforcing } FD \text{ on } t\}$	chase

3 Translating the relational algebra into Lucid

Lucid is a language that manipulates ω -dimensional streams. In other words, a Lucid program will manipulate unbounded streams that vary in an unbounded number of (named) dimensions. Translating the relational algebra into Lucid involves creating an appropriate data structure for relations and defining, for each relational algebra operator, the appropriate Lucid expression.

For the purposes of this work, two dimensions are always needed (more will be used for temporary computations): the *da* dimension, viewed horizontally, is used for attributes; the *dt* dimension, viewed vertically, is used for tuples. A relation scheme R becomes, in Lucid, a one-dimensional stream that varies in the *da* dimension, and its values are attribute names, which become Lucid strings. A relation instance r becomes, in Lucid, a two-dimensional stream that varies in the *da* and *dt* dimensions. A relation occurrence is simply a pair $\langle R, r \rangle$. The set of functional dependencies FD is represented as a three-dimensional stream that varies in the *da*, *dt* and *dh* dimensions. This last dimension, *h* for hand, designates the left hand side (*dh*= 0) or the right hand side (*dh*= 1) of an FD .

3.1 Translation

$\llbracket \langle R, r \rangle \rrbracket$	$= \langle R, r \rangle$
$\llbracket r \cup s \rrbracket$	$= \text{Union}(\llbracket r \rrbracket, \llbracket s \rrbracket)$
$\llbracket r \cap s \rrbracket$	$= \text{Inter}(\llbracket r \rrbracket, \llbracket s \rrbracket)$
$\llbracket r - s \rrbracket$	$= \text{Diff}(\llbracket r \rrbracket, \llbracket s \rrbracket)$
$\llbracket \sigma_F(r) \rrbracket$	$= \llbracket r \rrbracket \text{ wvr. dt } \llbracket F \rrbracket r$
$\llbracket \pi_X(r) \rrbracket$	$= \text{Proj}(\llbracket r \rrbracket, [X])$
$\llbracket \rho_{Y \leftarrow X}(r) \rrbracket$	$= \text{Ren}(\llbracket r \rrbracket, [X], [Y])$
$\llbracket r \bowtie s \rrbracket$	$= \text{Join}(\llbracket r \rrbracket, \llbracket s \rrbracket)$
$\llbracket \Omega_{FD}(r) \rrbracket$	$= \text{Chase}(\llbracket r \rrbracket, [FD])$
$\llbracket F_0 \vee F_1 \rrbracket r$	$= \llbracket F_0 \rrbracket r \text{ or } \llbracket F_1 \rrbracket r$
$\llbracket F_0 \wedge F_1 \rrbracket r$	$= \llbracket F_0 \rrbracket r \text{ and } \llbracket F_1 \rrbracket r$
$\llbracket \neg F \rrbracket r$	$= \text{not } (\llbracket F \rrbracket r)$
$\llbracket A_0 \theta A_1 \rrbracket r$	$= \llbracket A_0 \rrbracket r \theta \llbracket A_1 \rrbracket r$
$\llbracket A \theta a \rrbracket r$	$= \llbracket A \rrbracket r \theta a$
$\llbracket A \rrbracket r$	$= r @ . da (da \text{ asa. da } (R \text{ eq } "A"))$
$\llbracket A_0 \dots A_n \rrbracket$	$= "A_0" \text{ fby. da } \dots \text{ fby. da } "A_n" \text{ fby. da eod}$

3.2 Lucid operators for the relational algebra

The above translation requires a number of new Lucid functions, as well as general functions which can be used in situations other than the relational algebra. The former are given in this section, the latter in the next.

Union

```
Union(<R,r>,<S,s>) = <compatible(R,S),union.dt.da(r,s)>
```

Intersection

```
Inter(<R,r>,<S,s>) = <compatible(R,S),inter.dt.da(r,s)>
```

Difference

```
Diff(<R,r>,<S,s>) = <compatible(R,S),diff.dt.da(r,s)>
```

Projection

```
Proj(<R,r>,X) = <R,r> wvr.da in.da(R,X)
```

Renaming

```
Ren(<R,r>,X,Y) = <S,r>
where
  S = if iseod(first.da R) then eod
      elseif in.da(first.da R,X)
      then (Y @.da (da asa.da (first.da R eq X))) fby.da
           Ren(next.da R,X,Y))
      else first.da R fby Ren(next.da R,X,Y)
      fi
end
```

Natural join

```
Join(<R,r>,<S,s>) = <union.da(R,V), join>
where
  dimension d0
  join = append.da(r, Proj(<S,s0>,V))
  s0 = s1 wvr.d0 match
  s1 = s @.dt d0
  match = equal.d0(Proj(<S,s1>,T), Proj(<R,r>,T))
  T = inter.da(R,S)
  V = diff.da(R,S)
end
```

Chase

```
Chase(r,F) = s0 asa.d0 iseod(F)
where
  dimension d0,d1,d2
  s0 = r fby.d0 s1 asa.d1 iseod(t)
  t = s0 @.d1 T
  s1 = s0 fby.d1
      if not do_match then s1
      else s2 asa.d2 iseod(rhs)
```

```

s2 = s1 fby.d2 (if A=rhs then s3 else s1)
s3 = if iszeron(t) then s2
      elseif iszeron(s2) then t
      elseif s2 eq t then s2
      else error
do_match = match asa.d2 (match eq false) or iseod(lhs)
match = (s1 @.da lhs) eq (t @.da lhs)
lhs = first.dh F
rhs = first.dh next.dh F
end

```

Compatibility of two scheme dataflows

```
compatible(R,S)= if equal.da(R,S) then R else error
```

3.3 General Lucid operators

Equivalence of two dataflows

```

equal.D(f0,f1) = match asa.D
                  (match eq false) or (iseod f0 and iseod f1)
where
  match = f0 eq f1
end

```

Inclusion of a dataflow in another

```

In.D.E(f0,f1) = match asa.D (match eq false) or (iseod f0)
where
  dimension d0
  match = match1 asa.d0 (match1 eq false) or (iseod f0 and iseod f2)
  match1 = f0 egal.E f2
  f2 = f1 @.D d0
end.

in.D(f0,f1) = match asa.D (match eq false) or (iseod f0)
where
  dimension d0
  match = match1 asa.d0 (match1 eq false) or (iseod f0 and iseod f2)
  match1 = f0 eq f2
  f2 = f1 @.D d0
end.

```

The need for these two different operators comes from the fact that Lucid prevents us from abstracting more than one dimension when computing an operation on two streams.

Appending a dataflow to another

```

append.D(f0,f1) =
  if iseod f0 then f1
  else f0 fby.D append.D(next.D f0,f1)
  fi

```

Union of two sets as dataflows

```
union.D.E(r,s)=
```

```

if iseod(first.D s) then r
elseif In.D.E(first.D s,r) then union.D.E(next.D s,r)
else first.D s fby.D union.D.E(next.D s,r)
fi

```

Intersection of two sets as dataflows

```

inter.D.E(r,s)=
  if iseod(first.D r) then eod
  elseif In.D.E(first.D r,s)
  then first.D r fby.D inter.D.E(next.D r,s)
  else inter.D.E(next.D r,s)
fi

```

Difference of two sets as dataflows

```

diff.D.E(r,s)=
  if iseod(first.D r) then eod
  elseif in.D(first.D r,s) then diff.D.E(next.D r,s)
  else first.D r fby.D diff.D.E(next.D r,s)
fi

```

4 Translation examples

Given the preceding set of Lucid expressions and their respective semantic translation from RA, a set of examples using the previous Lucid operators are presented in the following. These are simple examples which relational algebra is capable of representing. The database used in the examples (Figure 3) is inspired from data concerning the British (Tudor) royal family [1].

Example 1: Generate a database instance that represents the members of the Tudor royal family that were alive in the reing of each sovereign:

$$\sigma_{(Birth < Death) \wedge (Death > From)}(\tau_1 \bowtie \tau_2)$$

The corresponding Lucid expression is the following:

```

s wvr.dt (val_birth < val_death) and (val_death > val_from)
where
  val_birth = s @.da (da asa.da S eq "birth")
  val_from = s @.da (da asa.da S eq "from")
  val_death = s @.da (da asa.da S eq "death")
  <S,s> = Join(<R1,r1>,<R2,r2>)
end

```

Example 2: Generate the instance that represents the name, year of end of reign and the year of death of each Tudor sovereign that abdicated or was dethroned:

$$\pi_{Name,To,Death}(\sigma_{(Death > From)}(\rho_{Name \leftarrow Sovereign}(\tau_1) \bowtie \tau_2))$$

The corresponding Lucid expression is the following:

```

Proj(<S,s1>,"name" fby "to" fby "death" fby eod))
where
  s1 = s wvr.dt val_death > val_to
  val_death = s @.da (da asa.da S eq "death")
  val_to = s @.da (da asa.da S eq "to")

```


reign (r_1)			persons (r_2)			
Sovereign	From	To	Name	Sex	Birth	Death
Henry VII	1485	1509	Henry VII	M	1447	1509
Henry VIII	1509	1547	Arthur	M	1480	1502
Edward VI	1547	1553	Margaret	F	1484	1541
Mary I	1553	1558	Henry VIII	M	1491	1547
Elizabeth I	1558	1603	Mary	F	1495	1533
			Mary I	F	1516	1558
			Elizabeth I	F	1533	1603
			Edward VI	M	1537	1553

fatherhood (r_3)		motherhood (r_4)	
Father	Child	Mother	Child
Edmund Tudor	Henry VII	Margaret Beaufort	Henry VII
Henry VII	Henry VII	Elizabeth York	Arthur
Henry VII	Arthur	Elizabeth York	Margaret
Henry VII	Margaret	Elizabeth York	Henry VIII
Henry VII	Mary	Elizabeth York	Mary
Henry VIII	Edward VI	Jane Seymour	Edward VI
Henry VIII	Mary I	Catherine	Mary I
Henry VIII	Elizabeth I	Anne Boleyn	Elizabeth I

Figure 3: Database on the royal (Tudor) British family

```

val_from = s @.da (da asa.da S eq "from")
R0 = Ren(R1,"sovereign" fby eod,"name" fby eod)
<S,s> = Join(<R0,r1>,<R2,r2>)
end

```

Example 3: Generate a relation instance that represents all the Tudor sovereigns and their respective children:

$$\pi_{Sovereign,Child}(\tau_1 \bowtie (\rho_{Sovereign \leftarrow Father}(\tau_3) \cup \rho_{Sovereign \leftarrow Mother}(\tau_4)))$$

The corresponding Lucid expression is the following:

```

Proj(<T,t>, "sovereign" fby "child" fby eod)
where
  <T,t> = Join(<R1,r1>,<S0,s0>)
  S0 = Ren(<r3,R3>,"father" fby eod,"sovereign" fby eod)
  R1 = Ren(<r4,R4>,"mother" fby eod,"sovereign" fby eod)
  s0 = Union(r3,r4)
end

```

5 The transitive Closure: A possibility

The relational algebra offers no possibility for general recursion or iteration. As a result, computations that require an unbounded number of steps cannot be effected unless, as for union and intersection, they are predefined.

A typical example of such a computation is the transitive closure of a binary relation. Given a binary relation r , computing its transitive closure r^+ implies iterating an unbounded number of times through the relation, augmenting it with the missing pairs each time, until the whole system stabilizes.

This process cannot be expressed in the standard relational algebra. However, Lucid's general iterative methods offer a simple solution. Consider an instance r of a binary relation over the scheme $\{Succ, Pred\}$. Its transitive closure is given by

```

TransitiveClosure(<R,r>) = inst asa.d0 equal.dt(s,next.d0 s)
where
  dimension d0
  <S,s> = inst
  inst = <R,r> fby.d0
        Union(inst, Proj(Join(Ren(inst,"succ" fby eod,"i1" fby eod),
                               Ren(inst,"pred" fby eod,"i1" fby eod)),
               "pred" fby "succ" fby eod))
end

```

As an example, we show the binary relation instance r_5 , representing the Tudor succession, along with its transitive closure.

r_5		TransitiveClosure(R_5, r_5)	
Pred	Succ	Pred	Succ
Henry VII	Henry VIII	Henry VII	Henry VIII
Henry VIII	Edward VI	Henry VIII	Edward VI
Edward VI	Mary I	Edward VI	Mary I
Mary I	Elizabeth I	Mary I	Elizabeth I
Henry VII	Edward VI	Henry VII	Edward VI
Henry VIII	Mary I	Henry VIII	Mary I
Edward VI	Elizabeth I	Edward VI	Elizabeth I
Henry VII	Mary I	Henry VII	Mary I
Henry VIII	Elizabeth I	Henry VIII	Elizabeth I

Figure 4: The Tudor succession and its transitive closure.

6 Conclusion

The relational algebra has been translated into Lucid, by supposing that a relation instance is a bi-dimensional stream varying over tuples and attributes. In addition, computations that require an unbounded number of iterations, including chasing functional dependencies in a relation, are easily solved in Lucid, because iteration over an arbitrary dimension is a built-in concept. In particular, the transitive closure of a binary relation has been defined.

Note also that since Lucid handles all dimensions orthogonally, the Lucid functions that we wrote are surprisingly simple.

The above work is completely independent of concepts such as time and space. Time has already been added to dataflow languages to form real-time languages such as LUSTRE and RLucid. Similarly, the DATALOG language has been augmented to become Temporal DATALOG. If these notions are all combined into a coherent whole, we have the potential for a unified real-time database/programming language.

References

- [1] Atzeni and DeAntonellis. *Relational Database Theory*. Benjamin Cummings publishing company, 1993.
- [2] E.F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):76–90, 1970.
- [3] Faustini and Jaggannathan. Multidimensional Problem Solving in Lucid. Technical report, SRI International, 1993.

-
- [4] Peter Gray. *Logic, Algebra and Databases*. Ellis Horwood Limited, 1984.
 - [5] Joey Paquet. Why the Universal Relation should not be Forgotten. Technical report, Department of Computer Science, Laval University, 1993.
 - [6] Jeffrey Ullman. *Principles of Database Systems*. Computer Science press, second edition, 1983.
 - [7] William W. Wadge. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.
 - [8] William W. Wadge. Higher Order Lucid. Technical report, Department of Computer Science, University of Victoria, 1992.
 - [9] Chao-Chin Yang. *Relational Databases*. Prentice-Hall, 1986.