THE INTENSIONAL RELATION

JOEY PAQUET
Département d’ informatique, Université Laval
Ste-Foy (Québec), Canada G1K 7P4
E-mail: Joey.Paquet@ift.ulaval.ca

and

JOHN PLAICE
E-mail: John.Plaice@ift.ulaval.ca

ABSTRACT

This paper includes an intensional overview of the universal relation in database theory. It is argued that all of the concepts therein can be intensionalized, and in so doing, can address several attacks made against the universal relation.

1. Introduction

The universal relation is a concept that is now commonly used to express the semantics of dependencies in relational databases.\textsuperscript{1,2,3,6,7} According to this idea, an interface is provided by a database to give the impression that all queries and all updates are made on a single relation. This universal relation is in fact the natural join of the relations that make up the database.

Before the introduction of the universal relation, assumptions about the real world had to be expressed using functional dependencies and multivalued dependencies. The former are fairly easy to discern and enumerate; however, the same does not hold for multivalued dependencies. The universal relation, when appropriate, allows the multivalued dependencies to be automatically inferred from a single join dependency.

Databases built using the universal relation assumption make several basic assumptions, presented below.

1. Each attribute $A$ is expected to play a single role. In particular, puns are not allowed. For example, attribute ADDRESS cannot refer to both SENDING ADDRESS and to RECEIVING ADDRESS.

2. Each combination $[X]$ of attributes is expected to have a unique meaning. For a given set $X$ of attributes, there may well be several relationships, but there should be one that is most basic.

3. All access paths to compute the connection on $X$ represent the same “flavor” of relationship among the attributes in $X$. This means that if several possible paths to compute $X$ exist, they all have the same basic meaning.
Much of the theoretical work in databases in the early eighties dealt with the problems of building databases with universal relation interfaces. In most of these systems, an actual universal relation is never really built. Rather, there are a number of base relations that participate in a join dependency, which defines the universal relation interface. As a result, a user never need worry about how exactly to navigate through a database.

Notwithstanding certain critiques (see \textsuperscript{4,5,9} for a fuller discussion), the universal relation is very useful for expressing the semantics of dependency theory. Nevertheless, it should be clear that the three basic assumptions described above cannot be applied in all situations.

Some problems can be resolved in some situations. Names can be lengthened to ensure that all attributes play a single rôle. But if a meta-database combining information from many different databases were created, the information therein might be of such a different nature that it would be difficult to create a universal relation.

What is important to note is that each of the three notions can be intensionalized. Suppose that there were a natural notion of possible world in databases. Then the universal relation assumption would be transformed into the intensional relation assumption as follows: "In each possible world, each of the three assumptions must be satisfied."

It turns out that the natural concept of possible world already exists: it is the concept of "maximal objects". These correspond to the largest objects in which one is willing to navigate automatically. These can be constructed automatically from the functional dependencies (essentially they ensure that there are no functional dependency cycles in a maximal object).

In fact, the only navigation that should be explicitly given is navigation between maximal objects. In fact, this was actually suggested in a 1983 paper\textsuperscript{6} in which a banking example uses two maximal objects: in one CUSTOMER means DEPOSITOR and in the other CUSTOMER means BORROWER. It is suggested that aliases be allowed to indicate in which maximal object the object is considered to lie.

The intensional relation is therefore the universal relation in which one can navigate from one maximal object to another. It is not just a technical trick: it should simplify many problems of interoperability between heterogeneous databases.

This paper is a first attempt to understand relational databases from an intensional point of view. It begins with a review of definitions of relational databases. It is then shown that the basic relational operators can all be simulated using Lucid, and that Lucid's iterative capacities allow such operations as the computation of the transitive closure of a binary relation. Finally, it is shown how the notion of maximal object can also be translated into the intensional
model, without changing any of the previous aspects.

2. Definitions

Definition 1. A relation scheme \( R \) is a pair \( (X; D) \), where \( X \) is a finite set of names, called attributes, \( \{A_0, A_1, \ldots, A_n\} \), and \( D \) is a function \( \{A_i \mapsto D_i\}_{i=0..n} \). Each \( D_i \), called the domain of \( A_i \), is a non-empty set of values. One can write, as needed, \( R \), \( R(X) \) or \( R(X; D) \). For attributes, one can write \( X \), \( A_0 A_1 \cdots A_n \) or \( \{A_0, A_1, \ldots, A_n\} \). The juxtaposition of two attribute sets means their union.

The domain of \( R \), written \( D(R) \), is \( \bigcup_{i=0..n} D_i \).

Definition 2. Let \( R(X; D) \) be a relation scheme, where \( X = A_0 A_1 \cdots A_n \).
A tuple \( t \) is a function \( \{A_i \mapsto d_i\}_{i=0..n} \), where \( d_i \in D(A_i) \). For \( Y \subseteq X \), \( t[Y] \) is the restriction of \( t \) to \( Y \).

Definition 3. Let \( R(X; D) \) be a relation scheme. A relation \( r \) over \( R \) is a countable set of tuples over \( R \). One can write \( r \), \( r(R) \) or \( r(X; D) \). The domain of \( r \), written \( D(r) \), is the same thing as \( D(R) \). The active domain of \( r \), written \( D_0(r) \), is the minimal subset of \( D(r) \) containing all values appearing in tuples contained in \( r \).

Definition 4. A database scheme \( \mathbf{R} \) is a set of schemes \( \{R_0, R_1, \ldots, R_m\} \).
One normally writes \( \mathbf{R} = (R_0 R_1 \ldots R_m) \). The domain of \( \mathbf{R} \), \( D(\mathbf{R}) \), is \( \bigcup_{j=0..m} D(R_j) \).

Definition 5. Let \( R(R_0 R_1 \ldots R_m) \) be a database scheme. A database \( r \) over \( \mathbf{R} \) is a function \( \{R_j \mapsto r_j\}_{j=0..m} \), where each \( r_j \) is a relation over \( R_j \). One can write \( r \) or \( r(\mathbf{R}) \). The domain of \( r \), written \( D(r) \), is the same thing as \( D(\mathbf{R}) \).

The active domain of \( r \), written \( D_0(r) \), is the minimal subset of \( D(r) \) containing all the values appearing in tuples contained in \( r \).

Definition 6. Let \( R(X; D) \) and \( R'(X'; D') \) be two relation schemes. These are said to be equivalent (written \( R \equiv S \)) if and only if \( X = X' \) and \( D = D' \).

Definition 7. Let \( \mathbf{R} \) be a database scheme. An integrity constraint \( \gamma \) over \( \mathbf{R} \) is a Boolean function defined over \( \mathbf{R} \).

The most common integrity constraints in relation database management systems are key dependencies and functional dependencies.

Definition 8. Let \( r(X; D) \) be a relation and \( \emptyset \neq K \subseteq X \). The relation \( r \) satisfies the key dependency over \( K \), written \( \text{key}(K) \), if and only if
\[
\forall t_0, t_1 \in r \ (t_0[K] = t_1[K] \Rightarrow t_0 = t_1).
\]
The attributes \( K \) are called the key of relation \( r \).

Definition 9. Let \( r(X; D) \) be a relation and \( \emptyset \neq Y, Z \subseteq X \). The relation \( r \) satisfies the functional dependency of \( Y \) on \( Z \), written \( Y \rightarrow Z \), if and only if
\[
\forall t_0, t_1 \in r \ (t_0[Z] = t_1[Z] \Rightarrow t_0[Y] = t_1[Y]).
\]
Definition 10. A database scheme with constraints is a pair \((R; \Gamma)\), where \(R\) is a database scheme and \(\Gamma\) is a set of integrity constraints over \(R\). The set of functional dependencies over \(\Gamma\) is written \(\mathcal{F}_\Gamma\).

Definition 11. Let \(R \in (R; \Gamma)\) be a relation scheme. The set of functional dependencies over \(R\) is written \(\mathcal{F}_R(R)\).

Definition 12. Let \(Z = \{\zeta_i\}_{i \in \mathbb{N}}\) be an countably infinite set of null values, or zeron. Let \(D\) be a domain. Then \(D_Z = D \cup Z \cup \{T\}\) is a domain augmented with zeron, representing unknown values. Each zeron is considered to be distinct. A partial order \(\preceq\) is added to \(D_Z\) such that \(d \preceq T\), \(d \preceq d\), \(\zeta_i \preceq d\), and \(\zeta_i \preceq \zeta_j\) if \(i \geq j\), for every \(d \in D\) and \(\zeta_i, \zeta_j \in Z\).

Definition 13. A relation \(r\) is called a tableau, written \(T\), if \(D_0(r)\) contains zeron.

Suppose that there is a functional dependency \(\text{Telephone} \rightarrow \text{Address}\). The insertion of another tuple representing another entity named Joan Blo with the same telephone as Joe Blo would mean that value \(z_0\) should be assigned to attribute \(\text{Address}\). So, the database would contain the two tuples:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Blo</td>
<td>Joe</td>
<td>(z_0)</td>
<td>555-1234</td>
</tr>
<tr>
<td></td>
<td>Joan</td>
<td>(z_0)</td>
<td>555-1234</td>
</tr>
</tbody>
</table>

If the address of either Joe or Joan is found, the two zeron will be replaced by the same value.

Definition 14. Let \(T\) be a tableau and let \(\mathcal{F}\) be the set of functional dependencies over \(T\). The application of \(\mathcal{F}\) to \(T\), written \(\text{CHASE}_\mathcal{F}(T)\), follows the following algorithm:

\[
\begin{align*}
T' & := T \\
\text{while (there exists } t_1, t_2 \in T' \text{ and } X \rightarrow A \in \mathcal{F} \\
& \text{such that } (t_1[X] = t_2[X] \text{ and } t_1[A] \neq t_2[A]) \\
& \text{do} \\
& \quad t_1[A] := \max(t_1[A], t_2[A]); \\
& \quad t_2[A] := \max(t_1[A], t_2[A]); \\
& \text{end do;}
\end{align*}
\]

\(\text{CHASE}_\mathcal{F}(T) := T'\)

Definition 15. Let \(r\) and \(s\) be two equivalent relations. Their union, intersection and difference are, respectively,

\[
\begin{align*}
\text{r } \cup \text{s} & \equiv \{t \mid t \in r \lor t \in s\}; \\
\text{r } \cup \text{s} & \equiv \{t \mid t \in r \land t \notin s\}; \\
\text{r } \cup \text{s} & \equiv \{t \mid t \in r \land t \notin s\}.
\end{align*}
\]
Definition 16. Let \( r(X; D) \) be a relation and \( f : X \rightarrow X' \) be a bijection renaming the attributes. The renaming of \( r \) by \( f \) is given by:
\[
\rho(f)(r) = \tau(X'; D \circ f^{-1}).
\]
The tuples do not change.

Definition 17. Let \( r(R) \) be a relation and \( X \subseteq R \). The projection of \( r \) over \( X \) is given by:
\[
\pi_X(r) = \{ t[X] \mid t \in r \}.
\]

Definition 18. Let \( r(A_0A_1 \ldots A_n) \) be a relation, \( \theta \in \{=, \neq, <, >, \geq, \leq\} \), and \( c \in D(r) \). Selection in \( r \) is given by:
\[
\sigma_{(A_0\theta c)}(r) = \{ t \in r \mid t[A_0] \theta c \},
\]
\[
\sigma_{(A_0 \theta A_1)}(r) = \{ t \in r \mid t[A_0] \theta t[A_1] \}.
\]

Definition 19. Let \( r(XY) \) and \( s(YZ) \) be two relations such that \( YX \cap XZ = X \). The natural join of \( r \) and \( s \), is given by:
\[
r \bowtie s = \{ t[YXZ] \mid t_1 \in r \land t_2 \in s \land t[XY] = t_1[XY] \land t[XZ] = t_2[XZ] \}.
\]

3. Relations in Lucid

This section will show that it is possible to simulate a relational database in Lucid. The basic idea is that a relation is a stream that varies in two dimensions, da for attributes and dt for tuples. Once this basic concept is understood, then all of the standard operators can quickly be programmed in Lucid.

Let \( R(A_0A_1 \ldots A_n) \) be a relation scheme. In Lucid, it is represented by a one-dimensional stream \( r \) varying in the da dimension (see Figure 1):
\[
A_0 \text{ fby. da } A_1 \text{ fby. da } \ldots \text{ fby. da } A_n \text{ fby. da eod}
\]
Now \( A_0A_1 \ldots A_n \) is actually a set, so there are many possible translations. To simplify the presentation, we will assume that attributes belong to some totally ordered set, and that any relation places those attributes in strictly increasing order. Each operator will be required to preserve this order.

Let \( r(R) \) be a relation. In Lucid, it is represented by a pair \(<\mathcal{R}, r>\), where \( \mathcal{R} \) is as above and \( r \) is a two-dimensional stream varying in the da and dt dimensions (see Figure 2). Clearly if this pair makes any sense, then the variation in the da dimension in \( r \) should correspond to that in \( \mathcal{R} \).

We use several general-purpose Lucid functions to present the relational operators.

The first example is \texttt{realign}, which is used to \texttt{turn} its argument \texttt{E}. The latter’s variation in dimension \( c \) is transformed into variation in dimension \( d \):
realign_c_d(E) = E 0_c #_d

The equal function is a general equality function that works on n-dimensional finite arrays. The argument D is a stream of dimensions, in which vary E0 and E1; D itself varies in dimension d.

   equal_d,D(E0,E1) = s asa_d iseod(D)
   where
   s = E0 eq E1 fby_d collection_D s;
   collection_c E = (iseod(E) or E) asa_c
   (iseod(E) or not E);

   end;

The behavior of equal is illustrated by Figure 3.

The in function is used to determine if element e can be found in stream E, varying in dimension d. The equality argument is a Boolean function used to compare e and elements of E:

   in_d(equality,e,E) = match asa_d (not match or iseod(E0))
   where
   match = equality(e,E);

   end;

The included function is used to determine if each of the elements in stream E0, varying in dimension d, can also be found in stream E1:

   included_d(equality,E0,E1) =
   match asa_d (not match or iseod(E0))
   where
Fig. 3. Behavior of equal function

dimension d0;
E11 = realign.d0,d(E1);
match = in.d0(equality,E0,E11)
end;
The union, inter and diff functions compute, respectively, the union, intersection and difference between two sets, encoded as streams varying in dimension d. Argument incl is a boolean function used to determine inclusion of an element of r in s:

{union, inter, diff}.d (incl, r, s) =
{append.d(r wvr.d not inrs, s),
 r wvr.d inrs,
 r wvr.d not inrs}

where

dimension d1;
inrs = incl.d1(r, s @.d1 #.d);
end;
The append function takes a finite stream r, varying in dimension in dimension d, and adds stream s at the end:

append.d(r,s) =

if iseqod(first.d r) then s
else r fby.d append.d(next.d r, s)
fi;

Now for the relational operators. As in the original definition, the definitions of Union, Inter and Diff are given together, as the body of the where clause is shared by all three:
\{\text{Union}(\langle R, r \rangle, \langle S, s \rangle), \\
\text{Inter}(\langle R, r \rangle, \langle S, s \rangle), \\
\text{Diff}(\langle R, r \rangle, \langle S, s \rangle)\} = \\
\{\text{equivalent, union\_dt}(\text{incl}, r, s), \\
\text{equivalent, inter\_dt}(\text{incl}, r, s), \\
\text{equivalent, diff\_dt}(\text{incl}, r, s)\}

\text{where}
\begin{align*}
dimension & \text{ dd;} \\
\text{equivalent} = & \begin{cases} 
\text{if} \text{ equal\_dd}(\text{da fby\_dd eod})(R, S) \\
\text{then } R \\
\text{else} \text{ error} \\
\text{fi}; \\
\text{incl} = \text{ in\_dt}(\text{equal\_dd}(\text{da fby\_dd eod}))
\end{cases}
\end{align*}
\text{end;}

\text{There are two kinds of selection. The first computes } \sigma_{(A \theta C)}(r): \\
\text{Sel1(test, A, c, } \langle R, r \rangle) = \langle RA, r \text{ wvr\_dt test}(Acol, c)\rangle \\
\text{where}
\begin{align*}
dimension & \text{ d0;} \\
RA = & \begin{cases} 
\text{if} \text{ in\_da(eq, A, R)} \\
\text{then } R \\
\text{else} \text{ error} \\
\text{fi}; \\
Acol = & r \text{ asa\_da (R eq A)};
\end{cases}
\end{align*}
\text{end;}

\text{The second computes } \sigma_{(A_0 \theta A_1)}(r): \\
\text{Sel2(test, A0, A1, } \langle R, r \rangle) = \langle RA, r \text{ wvr\_dt test}(Acol0, Acol1)\rangle \\
\text{where}
\begin{align*}
dimension & \text{ d0;} \\
RA = & \begin{cases} 
\text{if} \text{ in\_da(eq, A0, R)} \text{ and } \text{in\_da(eq, A1, R)} \\
\text{then } R \\
\text{else} \text{ error} \\
\text{fi}; \\
Acol0 = & r \text{ asa\_da (R eq A0)}; \\
Acol1 = & r \text{ asa\_da (R eq A1)};
\end{cases}
\end{align*}
\text{end;}

\text{Projection is computed using the Proj function:}
\text{Proj}(X, } \langle R, r \rangle) = \langle \text{inclXR}, r \rangle \text{ wvr\_da in\_d0(eq,R,X0) } \\
\text{where}
\begin{align*}
dimension & \text{ d0;} \\
\text{inclXR} = & \begin{cases} 
\text{if } \text{included\_da(eq,X,R)} \\
\text{then } R
\end{cases}
\end{align*}
else error
fi;
X0 = realign,d0.da(X);
end;
The renaming and join operators are defined similarly, although they are a bit
more complex since the order of the attributes must be maintained.

The translation of the relational operators is not just academic. There are
operations that cannot be effected in the relational algebra. One of the most
well-known is the transitive closure of a binary relation. In Lucid, because of
its general iteration capabilities, this problem is resolved simply:

\[
\text{Closure}(A1,A2,\langle R,r\rangle) =
\langle \text{S}, \text{s}\rangle \text{ asa} d0 \text{ equal} d1, \langle \text{da fby} d1 \text{ dt fby} d1 \text{ eod} \rangle
\]
\[
\text{(s, next} d0 \text{ s)}
\]

where
dimension d0,d1;
\[
\langle \text{S}, \text{s}\rangle = \langle \text{R}, \text{r}\rangle \text{ fby} d0
\]
\[
\text{proj(S, Join(Ren(A2,A,\langle S,s\rangle),Ren(A1,A,\langle S,s\rangle)))};
\]
end;

4. Functional dependencies in Lucid

Real databases are not just a collection of relations that happen to sit one
beside the other. All sorts of integrity constraints exist, and the various up-
dates must ensure that these constraints are respected as the database is being
modified. In this section, we show that functional dependencies can be encoded
in Lucid, and that the resulting chase algorithm can be programmed quite ele-
gantly in Lucid.

For the purposes of this presentation, we suppose that functional depen-
dencies do not pertain to attributes of different relations. It should be clear that this
restriction could be relaxed by adding another dimension, for relations.

A functional dependency varies in the dh dimension and the da dimension.
We explain with an example. Consider the dependency \(A_0A_1 \rightarrow B_0B_1B_2\). In
Lucid, it becomes:

\[
\langle \text{A0 fby} d0 \text{ A1 fby} d0 \text{ eod} \rangle \text{ fby} dh
\]
\[
\langle \text{B0 fby} d0 \text{ B1 fby} d0 \text{ B2 fby} eod \rangle \text{ fby} dh \text{ eod}
\]
So the dh dimension is used to distinguish the left- and right-hand side of the
dependency.

A set of functional dependencies is a 3-dimensional object. The df dimension
is used to express several dependencies. See Figure 4.

The fundamental algorithm used to ensure that a set of dependencies is re-
spected by a relation is called the chase algorithm. Chasing ensures that zeron
in a tableau can be instantiated according to the functional dependencies and also that there are no inconsistencies. The chase algorithm expresses itself elegantly in Lucid, because of its iterative capacities:

\[
\text{Chase}(F, <R,r>) = r_0 \text{ asa df iseod}(F)
\]

where

\[
\begin{align*}
r_0 &= r \text{ fby df onechase}(r_0); \\
lhs &= \#.da \text{ asa da (R eq first dh F)}; \\
lhs &= \#.da \text{ asa da (R eq first dh next dh F)}; \\
\text{onechase}(r) &= s \text{ asa d0 iseod}(t) \\
\end{align*}
\]

where

\[
\begin{align*}
dimension d0; \\
t &= \text{realign d0, dt}(r); \\
s &= r \text{ fby d0 if match}(s,t) \\
& \quad \text{then rewrite}(s,t) \\
& \quad \text{else s} \\
& \quad \text{fi}; \\
\end{align*}
\]

\[
\begin{align*}
\text{match}(s,t) &= m \text{ asa da (not m or iseod}(\text{lhs})) \\
\end{align*}
\]

where

\[
\begin{align*}
m &= (s \@.da \text{ lhs}) \text{ eq } (t \@.da \text{ lhs}); \\
\end{align*}
\]

\[
\begin{align*}
\text{rewrite}(s,t) &= p \text{ asa d1 iseod}(\text{rhst}) \\
\end{align*}
\]

where

\[
\begin{align*}
dimension d1; \\
\text{rhst} &= \text{realign d1, da rhst}; \\
p &= s \text{ fby d1 if } \#.da = \text{ rhst} \\
& \quad \text{then max}(p,t) \\
& \quad \text{else p} \\
& \quad \text{fi}; \\
\end{align*}
\]

end;
\text{max}(s,t) = \text{if } s \text{ eq } t \text{ then } s \\
\hspace{1cm} \text{elif iszero}(s) \text{ and iszero}(t) \\
\hspace{2cm} \text{then min_zero}(s,t) \\
\hspace{1cm} \text{elif iszero}(s) \text{ then } t \\
\hspace{1cm} \text{elif iszero}(t) \text{ then } s \\
\hspace{1cm} \text{else error} \\
\hspace{1cm} \text{fi;}

5. Universal relation

The universal relation was proposed during the early eighties to allow database users to make queries without having to explicitly state how navigation through the database's relations should take place. The basic idea is that the base relations are not visible, and that interaction between users and the database takes place through the attributes. For this idea to work, all the relations in the database must form a join dependency, i.e. they must act as if they are the projections of a single larger relation, and that the join of all these relations forms that larger relation.

In the real world, however, this restriction is just too strong. So Maier and Ullman, among others, developed the concept of 'maximal object'. Rather than try to force all the relations into a join dependency, smaller groups of relations, called \textit{maximal objects}, could be placed together. Within each maximal object, one can navigate freely without worrying about ambiguities; outside, one must be more careful.

A database can then consist of several maximal objects, which may share several relations among themselves. Queries can be made to the entire database, and only those maximal objects that include all of the attributes mentioned in that query are considered. If several tuple variables are used in a query language, then each variable may deal with a tuple in a different maximal object.

In Lucid, we represent a database that supposes the universal relation with a two-dimensional stream. The dm dimension is used to designate the maximal objects. Each maximal object is itself a stream of relations, varying in dimension dr. So every element in the database is a relation, itself a 2-dimensional stream (da and dt). See figure 5. In that figure, there is a third dimension, dv, used for queries, which is explained below.

We use a banking example.\textsuperscript{6} There are two maximal objects (see Figure 6):
\{C-ADR, C-ACC, ACC-BAL, ACC-BNK\}
and
\{C-ADR, C-L, L-AMT, L-BNK\}
Fig. 5. Query with maximal objects

Fig. 6. Banking example
where each pair is a base relation. The first one corresponds to a customer (with address) with a savings account (with balance) in a bank, while the second corresponds to a customer (with address) with a loan (with amount) from a bank. Notice that the C-ADR relation is shared by both maximal objects.

Consider in this banking example the query

\[
\text{retrieve(t.ACC)}
\]

\[
\text{where t.C = s.C and s.L = 4-326};
\]

The tuple variables range independently over the database. In this case, the only possibility is for \( t \) to range over the first maximal object and \( s \) to range over the second.

We can now present the algorithm for generating a relational algebra expression from such a query\(^6\):

Given query \( Q \) mentioning tuple variables \( t_1, t_2, \ldots, t_k \) and given maximal objects \( m_1, m_2, \ldots, m_q \), we convert \( Q \) to an algebraic expression \( E \) as follows:

1. For each \( t_i \), let \( X_i = \{B|t_i.B \text{ appears in } Q\} \). Let \( M_i \) be a set of maximal objects \( M_j \) such that \( X_i \subseteq U_j \), where \( U_j \) is the union of objects in \( M_j \).

2. For each maximal object \( m_i \), let \( J_i \) be the algebraic expression of all the relations on objects in \( m_i \).

3. For each tuple variable \( t_i \), construct the algebraic expression \( K_i \) to be the union of expressions \( J_j \) over all \( j \) such that \( m_j \) is in \( M_i \).

4. Let \( E = K_1 \times K_2 \times \cdots \times K_k \).

The use of the dv dimension now becomes apparent. To translate queries, each tuple variable is handled independently, by varying in dv. The rest of the translation is straightforward and resembles the translations for the relational operators. For reasons of space, we do not present them here.

Once we get used to the idea of adding dimensions to allow different levels of navigation, it is clear that the same tactic could be used for creating interoperating databases, with different levels of automatic navigation. These ideas merit further research.

6. Conclusions

We have shown that various aspects of relational database theory can be readily translated into an intensional framework. In most situations, the trick seems to be to add a new dimension. Moreover, the intensional approach, with its multiple dimensions, seems to be suitable to solving, in a general manner, several basic problems in relational database theory. These include computing the transitive closure of a binary relation and helping us understand the complexities around the universal relation.
Clearly, the research presented here is only preliminary and has not yet been developed into a full theory. Nevertheless, we think that these preliminary results are encouraging.

7. References


