Investigating A Relational ‘Compose’ Operator

David Livingstone
Dept of Computing, University of Northumbria

A new relational algebra operator is investigated and developed for use with database relations that hold data representing graphical structures. It is derived from the mathematical ‘Compose’ operator which creates a set of transitive mappings from its operand’s non-transitive mappings.

1. Introduction

Database relations are traditionally considered not very suitable for holding the data of hierarchical or other graph structures. This paper proposes a relational algebra Compose operator that will alleviate this problem when such structures are stored in a relation.

Precisely how a graph structure is represented in a relational variable (= relvar) of a database depends upon the relvar’s design. However a relvar representing a graph structure must at the very least contain 2 sets of attributes such that in each tuple, the 2 sets of attribute values represent two nodes and the fact that there is a mapping (or edge or arc, the terminology depends on the application area) from one node to the other. The relvar may or may not contain additional attributes.

The 2 node sets must contain the same positive number of attributes, and each attribute in one node set must have the same name and data type as the corresponding attribute in the other node set. A Compose operator must be able to handle any number of attributes in the node sets.

Any additional attributes – called non-mapping attributes – would be expected to contain data about the mapping between the two nodes and/or data about either or both of the nodes. However, the number and nature of the non-mapping attributes depends on the relvar’s design, so it cannot be assumed that non-mapping attributes are about mappings or nodes. A Compose operator must be able to function effectively regardless of how well or badly the relvar is designed.

This raises an important distinction, which is between on the one hand the design of a Compose operator as a formal, logical function that is incorporated into a formal, logical relational model, and on the other hand the use of that formal, logical relational model to design the set of relvars that are to constitute a relational database. Poor design of a relvar may inhibit the effective application of a Compose operator to that relvar. Nevertheless a well-designed Compose operator may ameliorate the limitations of a poorly-designed relvar.

For example, while the design of a specific relvar may be such that each tuple’s mapping is uni-directional (say a functional dependency), the use of 2 node sets to define mappings does not specify the direction of the mapping between the nodes. Therefore the formal specification of Compose should allow it to be applied in either
direction between the nodes. This could be useful with a relvar that holds the data of a management hierarchy (say); it would allow Compose to be used to move both up and down the hierarchy. Where mappings should only be used in one direction, it is the responsibility of the user to recognise this and only apply Compose in the valid direction.

As Date implies in [12], there are a great variety of different possible activities that one might wish to carry out as one follows through a chain of mappings, and the Compose operator should be able to help with as diverse a set of activities as possible. So it needs a clear conceptual focus but nevertheless be able to be put to a variety of uses.

This paper considers such an operator, deriving it from the ‘Compose’ operator of the mathematical theory of relations. Section 2 introduces the mathematical ‘Compose’ operator, and section 3 applies its principles to develop a simple relational algebra operator. Section 4 considers different kinds of graph structure and queries, and sections 5, 6, and 7 give examples of the use of Compose with these structures. Section 8 considers keys, and sections 9–12 consider terminating conditions, starting conditions, and more generalised approaches to mapping. Section 13 gives further examples. Sections 14 and 15 review the proposed final version of the operator. Lastly there is an appendix about implementation, and some references.

2. Relational Composition and Closure

In the relational theory of standard mathematics, two relations, say $R[a,b]$ and $S[c,d]$, can be composed together if $b$ and $c$ are of the same type. This would result in the relation $ANS[a,d]$. $ANS$ contains a set of pairs, each pair being a mapping from an $a$ attribute value to a $d$ attribute value. Each mapping arises in $ANS$ if:

- there is a mapping from the $a$ value to the $b$ value in $R$,
- a $c$ value equal to the $b$ value occurs in $S$,
- and there is a mapping from that $c$ value to the $d$ value in $S$.

Further composition of this type can be carried as far as required. For example, if ‘;’ is the composition operator, the composition

$$R[a,b] ; S[c,d] ; T[e,f] ; U[g,h] ; \ldots$$

can be carried out as long as $b$ and $c$ are of the same type, and so are $d$ and $e$, $f$ and $g$, and so on. The composition of $R$, $S$, $T$ and $U$ would result in relation $ANS[a,h]$ containing a set of mappings from $a$ attribute values to $h$ attribute values.

In standard mathematics, if a relation’s two attributes are both of the same type, a relation can be composed with itself. This composition can be repeated a number of times. It is sometimes done to obtain what is called a transitive closure. A transitive closure can be explained as follows. Suppose a relation $R$ contains the mapping ‘$a \rightarrow b$’, ‘$b \rightarrow c$’, and ‘$c \rightarrow d$’. Following these mappings, $a$ can be said to be linked to $c$ and $d$ transitively, i.e. via $b$ and $c$. $R$ is said to be a transitive relation if values that it relates transitively, it also relates in transitively, i.e. by the inclusion of the direct mappings (in the above example, ‘$a \rightarrow c$’, ‘$a \rightarrow d$’ and ‘$b \rightarrow d$’). A transitive closure of a relation is achieved by adding just those mappings to it which turn it into a transitive relation. These additional mappings can be obtained by composing a relation with itself, composing the result with the original relation, composing the result of that with the original relation, and so on until an empty result is obtained;
then a distributed union of all the results with the original relation yields the transitive closure of the relation.

The proposed database relational algebra Compose operator is based on the latter idea of \textit{composing a relation with itself as many times as desired}. This focus for the operator is for the following reasons:

- The composition of multiple relations can be done with Natural or Generalised Join operators (with Renaming where necessary) so no new operator is necessary.
- The composition of a single relation multiple times cannot easily be done without a new operator:
  - Sometimes the number of times that composition must be applied depends on the results obtained from each composition, say when a particular terminating condition or transitive closure is required. In this case an expression comprising a set number of join operations cannot achieve the required result.
  - Sometimes the compositions need to include calculations involving attribute values. Join operations are insufficient for this, and use of the \texttt{Extend} operator is precluded when it cannot be incorporated within a fixed join expression.

Hence the starting point for investigation is:

1. The mapping between nodes represented in each tuple is taken to be a non-transitive link in a graph structure.
2. The purpose of a Compose operator is to generate a relational value result of the same type as the operand, whose tuples each represent transitive links between nodes.
3. Ancillary related functionality may be built into the \texttt{Compose} operator if it naturally integrates with the creation of transitive links.

A consequence of this is that if a relvar contains a tuple that represents a non-transitive mapping between 2 nodes, and also contains 2 or more tuples whose mappings together define a transitive mapping between the same 2 nodes, then the graph structure has 2 distinct links between those 2 nodes. However formally \texttt{Compose} cannot treat the mappings of the transitive link differently to the non-transitive mapping. All mappings are treated identically. This consequence can be important in certain graph structures, e.g. cyclic structures.

### 3. Relational Databases : a Basic ‘Compose’ Operator

A basic version of a compose operator is first considered. Let it be written as

\[ R \text{ Compose} [ x; y ] \]

where:

- \( R \) is a name or expression that evaluates to a relational value (= relvalue).
- \( x \) and \( y \) are the names of 2 attributes in that relvalue that represent nodes between which mapping occurs. The mapping is always regarded as going from the first to the second attribute, i.e. the mapping is \( x \rightarrow y \) in this case. \( x \) is called the source attribute of the mapping and \( y \) is called the
target attribute of the mapping.
   (In general, x and y represent sets of attributes. However for
   convenience, sets of single attributes will henceforth be used to represent
   nodes).

- There could be additional non-mapping attributes in R that are not
  referenced in the above invocation of Compose. In general a set of non-
  mapping attributes must be assumed, but for the moment this set is assumed
  to be empty.

To execute a single composition

\[
R \; \text{Compose} \{ x; y \}
\]

the following expression is executed:

\[
R \; \text{Join} \{ y \} \; ( R \; \text{Rename} \{ y \rightarrow x; z \rightarrow y \} \) \; \text{Project} \{ y \} \; \text{Rename} \{ y \rightarrow z \}
\]

R is naturally joined with a copy of itself. However to enable Join to compare the
targets of the mappings in the left-hand instance of R (attribute y) with the sources of
the mappings in the right-hand instance of R (attribute x) the 2 attributes must have
the same name; therefore the right-hand instance of R has its x attribute renamed y.
Because there cannot be 2 attributes with the same name in the right-hand instance of
R, the attribute already called y (the target attribute) is renamed something else, z in
this case. The Join result contains one copy of attribute y, the intermediate node in
the mapping from x to the target now called z. Therefore y is projected out. Finally
the target attribute z in the result is renamed y in order to avoid giving the result a
random attribute name and to ensure that the result has the same type as R.

Suppose R in fact has a non-empty set of non-mapping attributes. Compose is
extended to cope with this as follows. Its result is defined to include the set of non-
mapping attributes, with their values being taken from either the left or the right-hand
instance of R. In terms of the above expression, the non-mapping attributes from
either the left or right instance of R are projected out before the Join. To specify
which non-mapping attributes are to appear in the result, an extra [ or ] bracket is used
thus:

\[
R \; \text{Compose}[[ x; y ]] \quad \text{or} \quad R \; \text{Compose} [ x; y ]
\]

to indicate the left and right-hand instance of R respectively.

Note that Compose is defined as a pseudo monadic; i.e. the user only provides one
operand, but two copies of it are used in the operation.

The above is a single composition, but typically a succession of composition
operations is required rather than just one. To achieve this, the relvalue which is the
result of the first composition is composed with the original operand, the result of that
is composed again with the original operand in a third iteration, and so on until the
required number of compositions has been achieved.

The Compose operator must manage the succession of compositions. This requires
that the Compose operator be able to take a parameter to specify the number of
successive compositions. The parameter should cope with two options:

1. Repeat the composition until an empty result is achieved, i.e. a transitive closure is
   obtained.

---

1 So for example, if the non-mapping attributes in the operand refer to (say) the source node, the non-
mapping attributes in the result also refer to the source node.
2. Repeat the operation a specified number of times. (An empty result might be achieved before the specified number of closures were completed. This need not result in an error since composing an empty result relation with the relational operand will yield another empty result, and so on until the required number of compositions have been completed. Of course, since it makes no difference to the result, the software need not actually continue with unnecessary compositions).

To express repeated compositions, the notation could be extended as follows:

\[ R \text{ Compose}[ x; y ] \text{ By[ n ]} \]

where \( n \) is the number of compositions, or ‘+’ if a transitive closure is to be derived. If the \( \text{By} \) clause is omitted, one compose operation is executed by default.

It is proposed that the result of applying ‘Compose’ to its operand consists of the distributed union of all the relvalues produced by the required composition iterations, these being determined by the parameter values passed to \text{Compose}.

This is because it is considered that a typical use of \text{Compose} would be to derive a wide variety of results containing mappings of various lengths of 2 or more. It is not thought that a typical use would be to generate a transitive closure – this would be a special case. It is easy to union the result with the operand to obtain a transitive closure if required; to remove the operand from a transitive closure result creates extra work which can only be avoided with an effective query optimiser.

Sometimes the reason for doing a composition is to obtain ‘route-based’ data; i.e. to obtain data that is derived from attribute values encountered in following along the mappings. To achieve this, let there be an additional (optional) parameter which specifies the calculations and those non-mapping attribute(s) to which the results of the calculations are to be assigned. The following example illustrates the idea:

\[ R \text{ Compose}[ x; y ] \text{ With[ c \leftarrow [c + 1 ; q \leftarrow [q \times q ] \text{ By[ 2 ]} \right.} \]

In this example the \text{With} parameter specifies 2 calculations and the non-mapping attributes to receive the results. \( \leftarrow \) assigns the value of the expression on its right to the attribute named on its left. Constant and/or attribute values may appear in the expression. Since \text{Compose} is pseudo monadic, it uses 2 identical copies of the operand; so both copies have the same set of attribute names. To distinguish attributes of the same name in the left and right-hand copies, a left-hand attribute name is preceded by a left-hand bracket, [, and a right-hand attribute name is preceded by a right-hand bracket, ].

The example shows attribute \( c \) incremented by one during each composition - this could correspond to adding up the levels of a hierarchy that were passed through - and result attribute \( q \) receiving the product of \( q \) in the left and right hand attributes - this could correspond to calculating the number of component parts that were required by a product in a Bill of Materials (= BOM).

The values of non-mapping attributes that are determined in the \text{With} parameter overwrite the values of these attributes that would otherwise be taken from the left or right-hand instance of the operand.

If the values of some non-mapping attributes are to be taken from the left-hand instance of the operand and some from the right-hand instance, then either the left or

---

\(^2\) In standard mathematical terminology, a relation name is annotated with ‘+’ to indicate the relational value which is its transitive closure.
right instance should be chosen and the remaining non-mapping attributes should have their value determined by the With parameter.

The With parameter is essentially the same in nature as the parameter of the Extend operator, and so should have the same syntax. It has a corresponding effect, except that:

- Extend extends the relation, whereas Compose does not since the result must be union compatible with the operand;
- Extend’s parameter refers to only one operand; but Compose’s may refer to 2 operands that have the same set of attribute names, which then have to be distinguished in the expression.

Since a crucial aspect of each tuple’s data is the mapping it represents, the attributes comprising the key can often be expected to include those forming the 2 nodes of the mapping. It will often be useful in future sections to assume this in order to consider problems of duplicate mappings.

However such a key is not mandatory. Where some other key is used, concerns about duplicate mappings can be solved by other means. Since they can be many and varied, only the use of keys is considered in this document.

4. What Kind of Graph Structures and Queries Should a Compose Operator be Able to Handle?

In essence there are three kinds of graph structure:

1. A tree structure; i.e. every node has exactly one parent node, except the root node which has none.
2. A Directed Acyclic Graph (= DAG); i.e. every node has zero, one or more parent nodes and zero, one or more child nodes, but never refers to itself either directly or indirectly.
3. A Directed Cyclic Graph (= DCG); i.e. every node refers to any other node(s) in the graph, including itself, directly or indirectly.

The requirement for a formal, general purpose Compose operator means that it should be able to handle all 3 kinds.

A succession of compositions can terminate in a transitive closure in the first two cases but not with a DCG, where an infinite succession of compositions can arise. Thus suitable terminating conditions must be used with a DCG.

A relation may – and often will – hold more than one graph structure. For generality Compose must be able to handle a set of graph structures in a relation. It is possible for all 3 kinds of graph structure to appear in the same relation.

The kinds of graph structure can affect the different kinds of query for which one might expect a Compose operator to be useful. Reference [10] provides the following useful categorisation of queries:

1. Cases where a single path is involved. This can be sub-divided as follows:
   - Reachability queries. For example, starting at node A, can one reach node B? Or what nodes can one reach from A?
The enumeration of a path and its properties. For example, how many direct mappings there are in a path from $A$ to $B$? How far is it from $A$ to $B$?

2. Cases where there are multiple paths between two given nodes; this is termed reconvergent in [10]. This case can also be sub-divided into two:
   - The enumeration of each individual path and its properties. For example, the classic BOM problem of how many components of each part type are needed to make product $A$?
   - The aggregation over all the paths and their properties, i.e. taking the set of paths as a whole. Often this refers to finding which path in the set is optimal in some sense. For example, which is the longest path from $A$ to $B$ - i.e. the critical path problem in a PERT network? What is the shortest route from city $A$ to city $B$?

Queries could also be a combinations of these. For example, what is the shortest route from city $A$ to city $B$ and through which other towns does it go?

Another categorisation is into extremal and non-extremal paths. ‘Extremal’ is essentially another term for optimisation, and can be of two kinds. The first corresponds to the reachability case above - e.g. how far from node $A$ can one get? The second corresponds to aggregation reconvergent cases where an optimum is sought. The non-extremal cases are simply the remainder not mentioned in the above categorisation.

Carre’s Path Algebra [11], also exploited by Ioannidis & Ramakrishnan [5] and Cruz & Norvell [9] is very useful for showing how to approach different kinds of query.

5. Examples of the Application of ‘Compose’ on Relvalues Representing Tree Structures

Consider the following relvar $R$, together with a graphical representation of the mappings held in it, which for convenience is considered to hold a BOM for a set of two products :-

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>g</td>
<td>3</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>p</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>q</td>
<td>4</td>
</tr>
</tbody>
</table>

```
 a
  |   |
 b   c
  |   |
  d   e  f
  |   |
  g
  |
 m
  |   |
 n   o
  |   |
  p   q
  |   |
```

Page 7 of 28
Then the single composition

\[ R \text{ Compose}[ \text{Super-Component} ; \text{Sub-Component} ] \]

\[ \text{With[ Quantity } \leftarrow \text{ [Quantity } \times \text{ [Quantity } \right] \]

would yield the following result relvalue - for convenience let it be called \textit{Ext1} - which corresponds to the extra (emphasised) mappings on the diagram:

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>a</td>
<td>e</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>f</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>g</td>
<td>15</td>
</tr>
<tr>
<td>m</td>
<td>p</td>
<td>6</td>
</tr>
<tr>
<td>m</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>

All the mappings represented in this relvalue are of length two, i.e. map a super-component to a sub-component via a single intermediary component. Note that because there are two trees represented in the relation, the composition applies to both of them. Note also that it is not simply the mappings from the top of the tree that are composed with their successor mappings, but all instances where a mapping has a successor mapping that can be composed with it.

A second composition (which would now be of \textit{Ext1} as the left-hand operand with \textit{R} as the right-hand operand) would yield the following relvalue \textit{Ext2}:

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
<td>30</td>
</tr>
</tbody>
</table>
This is just one mapping of length three that can be produced from the original relvar \( R \).

If a further composition were carried out (i.e. of \( Ext2 \) with \( R \)), the result would be an empty relvalue, as there are no length four mappings.

This illustrates that an extra iteration is required to find out if closure has been achieved, the iteration which yields an empty result.

From mathematical theory, a transitive closure of \( R \), written \( R^+ \), would be the distributed union of the original operand and the relvalues produced by each iteration of \textit{Compose}:

\[
R^+ \equiv R \text{ Union} \ Ext1 \text{ Union} \ Ext2
\]
\[
\equiv R \text{ Union} ( \{ \{ Ext1 \} \{ Ext2 \} \} \text{ Dist}\text{[ Union ]})
\]
\[
\equiv \{ \{ R \} \{ Ext1 \} \{ Ext2 \} \} \text{ Dist}[ \text{ Union]}
\]

However applying \textit{Compose} with the parameter \texttt{By[ + ]} to \( R \) would only yield

\( Ext1 \text{ Union} Ext2 \) or \( \{ \{ Ext1 \} \{ Ext2 \} \} \text{ Dist}[ \text{ Union]}
\]
and this result would have to be unioned with \( R \) to achieve \( R^+ \).

6. Examples of the Application of ‘Compose’ on Relvalues Representing Directed Acyclic Graphs

Consider the following relvar \( R2 \), together with a graphical representation of the mappings held in it, which for convenience is also considered to hold a BOM for a set of two different products :

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>g</td>
<td>8</td>
</tr>
<tr>
<td>e</td>
<td>h</td>
<td>6</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>o</td>
<td>5</td>
</tr>
<tr>
<td>n</td>
<td>o</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that component \( e \) is required by both component \( b \) and component \( c \). Similarly component \( o \) is required by both component \( n \) and component \( m \). Thus the graph structure is not a tree, although it still does not include a loop or cyclic structure.
Let the same single composition as before be applied to \( R2 \):

\[
    R2 \text{ Compose}[ \text{Super-Component} ; \text{Sub-Component} ]
\]

With[ Quantity \leftarrow [\text{Quantity} \times [\text{Quantity} \times [\text{Quantity} \times [\text{Quantity} \times [\text{Quantity}]])]
\]

It yields two transitive mappings ‘\( a \rightarrow e \)’ (one via \( b \) and one via \( c \)). If the quantities of components required are being calculated as part of the composition, the 2 tuples holding the mappings will contain different quantities. The first iteration also yields another mapping ‘\( m \rightarrow o \)’ (which is transitive via \( n \)); the quantity of components, if calculated, will be different to that in the tuple of the direct mapping ‘\( m \rightarrow o \)’.

Were it the case that the tuples of the transitive mappings contained the same quantity rather than different quantities, duplicate tuples would result, causing a problem.

The second \textbf{Compose} iteration should result in two transitive mappings of ‘\( a \rightarrow g \)’ and ‘\( a \rightarrow h \)’, due to the 2 mappings ‘\( a \rightarrow e \)’. There are no new mappings arising from product \( m \).

If a further composition were carried out, the result would be an empty relvalue, as there are no length four mappings. Thus 2 iterations are enough to achieve the additional mappings within a transitive closure.

The question arises as to how to deal with duplicate mappings. The question and its solution must be formulated in relational terms, because it is a \textit{relational} operand that the \textbf{Compose} operator is applied to, not a graph structure \textit{per se}. The graph structure is represented by the relvalue that is the operand.

The proposed solution is postponed to section 8 in order to take account of DCG structures.

### 7. Examples of the Application of ‘Compose’ on Relvalues Representing Directed Cyclic Graphs

Consider the following relvar \( R3 \), together with a graphical representation of the mappings held in it:

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>4</td>
</tr>
</tbody>
</table>

If the relvar were to represent a BOM, it would contain two errors, namely the 2 loops.

Let the same single composition as before be applied to \( R3 \):
It yields mappings of length two:

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>25</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>35</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>12</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>28</td>
</tr>
</tbody>
</table>

The shaded tuples are ones that we presumably would not want, because they duplicate identical mappings of a shorter length, i.e. of length one. They all arise because the loop ‘a → a’ has been traversed, ‘padding out’ the original mappings with this extra but superfluous mapping. Furthermore, if we leave them in, following iterations will give us yet more instances as ‘a → a’ is continually traversed. Let us therefore assume they are removed from the result before we move on to the next iteration.

The second iteration yields mappings of length three:

<table>
<thead>
<tr>
<th>Super-Component</th>
<th>Sub-Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>60</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>84</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>74</td>
</tr>
</tbody>
</table>

The shaded tuple of mapping ‘b → a’ corresponds to the shaded tuples in the previous iteration. We presumably would not want it for the same reason; it arises from the same cause as before. Again we assume it is removed before the next iteration.

The hatched tuples represent tuples that presumably we want in the result, because they are genuine mappings. They represent mappings that go round the loop that consists of three mappings. As they are self-mapping tuples, they will cause the same problems as the (direct) ‘a → a’ mapping in future iterations. They also include another ‘a → a’ mapping, a transitive mapping, which is problematic.

\(^3\) However they are unique tuples, since the ‘Quantity’ attribute value differs.
The third iteration yields mappings of length four. Because of the loops, presumably no mappings of this length are desirable, even the mapping ‘\(a \rightarrow d\)’, since it merely pads out the ‘\(a \rightarrow d\)’ mapping of length one via a traversal of the loop round \(b\) and \(c\).

Thus 2 iterations are enough to achieve the equivalent of a transitive closure.

The question arises as to how eliminate the unwanted tuples. This is considered in the next section.

### 8. Duplicate Mappings and the Use of Keys

Although the concepts underpinning \textit{Compose} relate to mappings in a graphical structure, \textit{Compose} is a relational operator, and its operands and results are relvalues. So it is the attributes and tuples of the relvalues that are expressly manipulated by \textit{Compose}. The mappings are interpretations of the data in the relvalues. Therefore \textit{Compose} has to use relational concepts to eliminate the tuples of unwanted mappings.

In a database relation, it is a requirement that a Key has a unique value in each tuple. Therefore Keys are considered as the general solution method for eliminating tuples that are unwanted because they hold unwanted mappings.

In order to exploit the use of keys, firstly \textit{Compose} is extended to include an optional parameter for specifying the Key to be applicable in the result relvalue. Secondly \textit{Compose} is defined such that any result tuple that it generates whose key value is duplicated by another tuple that it has \textit{already} generated in an \textit{earlier} iteration, is immediately discarded and not included in the result.

In order to clarify the use of Keys for this purpose, a Key is assumed to comprise all the mapping attributes (i.e. those that define the 2 nodes of the mappings) – let this be called a ‘Mapping Key’. Sometimes additional non-mapping attribute(s) may be incorporated into it to form an ‘Augmented Mapping Key’; for example, a calculated attribute(s) could be included if it is known that it will always contain a different value(s) for each instance of a duplicated Mapping Key.

In practice relvar designs might not use Mapping Keys. They might use Keys incorporating only one node or no nodes at all. It is not possible to consider all the design possibilities. It is merely considered that they must provide the same logical effect as the Mapping Key design strategies considered here.

It is useful to summarise the duplicate tuples arising from composition in the 3 graph structure examples:

- The tree structure example, relvar \(R\), shows that the longer mappings generated by \textit{Compose} do not include duplicate mappings.
- The DAG example, relvar \(R2\), shows that the longer mappings generated by \textit{Compose} can include duplicates of the original graph structure plus duplicates of other longer mappings.
- The DCG example, relvar \(R3\), shows that the longer mappings generated by \textit{Compose} can not only include duplicates of the original graph structure plus duplicates of other longer mappings, but also potentially infinite loops.

\footnote{For simplicity and convenience, multiple Keys are not considered. However in principle the option should allow a set of keys to be specified, which are all treated in the same way.}
Consider now the potential role of Keys when applying the Compose operator to each of the 3 examples:

**Example ‘R’**
Because duplicate mappings can never arise, the set of key values of a Mapping Key in the result are automatically unique and form a disjoint set to the set of key values in the operand. Therefore this case does not require that Compose include an optional parameter to specify keys for its result.

**Example ‘R2’**
Duplicate mappings do arise in this example. The Mapping Key (Super-Component, Sub-Component) is assumed to be the Key of operand R2, as it is the obvious key. Consider each iteration in turn:

- Result of first iteration.
  This yields 2 tuples with the same mapping ‘a → e’. If the Mapping Key is also the key of the result, there are 2 tuples with the same key value. There is only an arbitrary basis to discard one of the tuples, but more importantly both tuples are logically required in the result. Therefore an error should result, unless the result is specified to have an Augmented Mapping Key that makes each result tuple’s key value unique.

  The obvious Augmented Mapping Key is (Super-Component, Sub-Component, Quantity). Since R2 holds a BOM, the composition should calculate appropriate values for attribute ‘Quantity’ in the result. The nature of a BOM means that as mappings lengthen, then the corresponding ‘Quantity’ values will monotonically increase. However this is insufficient in itself to guarantee that the Augmented Mapping Key will have a unique value in every result tuple generated. Therefore unless specialist knowledge of R2 guarantees it, the Key will need to be augmented with a 4th attribute; this in turn requires that the operand is an extension of R2 such that the 4th attribute contains appropriate values, and that the composition includes an appropriate calculation of values for that 4th attribute in the result.

  The iteration also generates the result tuple with the mapping ‘m → o’. This tuple/mapping is unique in the result but also appears in R2; were the result to be unioned with R2 in order to obtain a transitive closure, the same duplicate tuple problem would arise. The result’s and the operand’s key values must form disjoint sets to obtain a transitive closure. The solution would be to give R2 a Key that corresponds to the result’s Key.

- Result of second iteration.
  This yields 2 tuples with the mapping ‘a → g’ and two with ‘a → h’. The problem and solution are a repeat of those of the first iteration for mapping ‘a → e’.

**Example ‘R3’**
Duplicate mappings again arise, as does the possibility of an infinite sequence of iterations. The obvious Mapping Key (Super-Component, Sub-Component) is again assumed to be the Key of operand R3. Consider each iteration in turn:

- Result of first iteration.
  The ‘undesirable’ shaded tuples in the result have key values that also appear in the operand, although their ‘Quantity’ attribute value differs. Therefore an ‘undesirable’ result tuple that has been generated can be discarded if its key value
is duplicated by another tuple that is either in the operand or in the result of an earlier iteration.

- Result of second iteration.
  As regards the ‘undesirable’ shaded tuple, the same applies as in the first iteration.
  It may be thought that, since a complete Compose result is a distributed union of the individual iteration results, duplicate tuples appearing in the second and subsequent iteration results will automatically be removed anyway. However this is not necessarily the case, as a union only treats tuples as duplicates if all the attribute values in both tuples are the same. In this case, only the Mapping Key attributes are certain to have identical attribute values; any non-mapping attributes may contain different values; but it is the tuples with duplicate key values that are to be discarded, regardless of the non-mapping attribute values.
  Regarding the hatched tuples, they are all retained as required, except for the tuple holding the ‘a \rightarrow a’ mapping which does have a duplicate key value and so is discarded. There is no way of treating the transitive ‘a \rightarrow a’ mapping differently from the direct ‘a \rightarrow a’ mapping. In any invocation of Compose, all the data in the operand must be treated in the same way.

Use of the Mapping Key allows all loops to be treated in an equivalent fashion, regardless of the number of mappings in a loop.

**Conclusions**

Two orthogonal considerations arise with the use of keys:

1. A Mapping Key may or may not need to be augmented with non-mapping attributes, depending on the mappings that are to appear in the tuples of the result.
2. When discarding a duplicate tuple on the basis of key values, it has to be decided whether the pre-existing tuples it is compared with are limited to those generated in a previous iteration, or whether they should additionally include the tuples in the operand. This depends on whether or not the key values of the result relvalue need to form a disjoint set to those of the operand.

**Key Parameter**

To handle keys, it is proposed that Compose be extended by optionally allowing either of the following parameter options:

- **KeyResult[ Name <-- AttributeName-Set ]** specifies that the result relvalue is to be given a key named ‘Name’ that comprises the set of attributes named in ‘AttributeName-Set’; when a result tuple is being considered as a potential duplicate, its key value is compared to that of tuples generated in previous iterations of the composition.

- **KeyBoth[ Name <-- AttributeName-Set ]** specifies that the result relvalue is to be given a key named ‘Name’ that comprises the set of attributes named in ‘AttributeName-Set’; when a result tuple is being considered as a potential duplicate, its key value is compared to that of tuples generated in previous iterations of the composition and compared to key values in the operand. The operand may not actually have a key comprising the ‘AttributeName-Set’ of attributes, but nevertheless these are the attributes to be used in the comparison with operand tuples.
Thus for the $R_2$ example (ignoring any By parameter), the Compose invocation might be:

\[
R_2 \text{ Compose} \left[ \text{Super-Component} \ ; \ \text{Sub-Component} \right] \\
\text{With} \left[ \text{Quantity} \leftarrow \text{[Quantity} \times \text{[Quantity]} \right] \\
\text{KeyResult} \left[ \text{Aug} \leftarrow \text{Super-Component}, \text{Sub-Component}, \text{Quantity} \right]
\]

For the $R_3$ example (ignoring any By parameter), the Compose invocation might be:

\[
R_3 \text{ Compose} \left[ \text{Super-Component} \ ; \ \text{Sub-Component} \right] \\
\text{With} \left[ \text{Quantity} \leftarrow \text{[Quantity} \times \text{[Quantity]} \right] \\
\text{KeyBoth} \left[ \text{NonAug} \leftarrow \text{Super-Component}, \text{Sub-Component} \right]
\]

If no Key parameter is given, the operand’s existing set of keys continue to be applied and appear in the result. If a Key parameter is given, then it will replace in the result any keys in the operand whose attributes form a proper subset of the parameter key.

It is important that the user understands the kind of graphical structure represented by a relation, so that when invoking Compose on it, they are able to choose the appropriate Key parameter to ensure that Compose is applied appropriately to that kind of structure.

9. A Summary of Questions Arising

The investigations so far give rise to 3 questions:

1. Is it sufficient that terminating conditions are specified solely by:
   - Specifying the number of iterations to be carried out?
   - Achieving a transitive closure (or its equivalent) by continuing until an iteration generates an empty relvalue?

   Would additional termination conditions be useful?

   Suppose the composition results of interest were those that contained transitive mappings of specific lengths, i.e. were built from $x$ direct mappings, where $x$ is a parameter input to Compose?

   What about compositions aimed at discovering certain kinds of nodes in the graph or paths through the graph relative to some initial nodes? Such compositions would need the ability to apply conditions after each iteration, with a view to ceasing composition after the conditions concerned had been met.

2. Is it sufficient that Compose be pseudo monadic, with one operand being provided and then used twice? Is there a case for allowing Compose to be explicitly dyadic, with 2 different operands?

   For example, suppose in a composition on a BOM, one is interested in only one or a few products and only wants compositions with respect to them? Typically it is better to hold a BOM for all one’s products in a single relvar, not only to avoid the inconvenience of having one relvar per product, but also because many components are likely to be used on many products and these components would be replicated over many relvars. Therefore what is required is the ability to Compose a relation of one or a few products – the ‘left-hand’ operand - with the complete BOM relvar – the ‘right-hand’ operand.

3. The Compose operator described above uses ‘=’ comparisons of attribute values. Should other comparisons be available as well?
For example, a relvar representing an airline timetable effectively represents mappings between nodes that are flights between airports, together with the departure and arrival times of the flights. Someone looking to fly between Britain and New Zealand (say) is looking for a sequence of connecting flights, i.e. a sequence of mappings. Therefore a **Compose** operation aimed at finding such flights needs to not only match intermediate nodes (i.e. airports) with an \( \text{“}=\text{”} \) comparison but also to match them with a departure time that is later than an arrival time\(^5\).

These questions are considered next.

10. Terminating Conditions

It is proposed that, in addition to transitive closure (supported by the use of Keys), there be 2 sorts of terminating conditions:

1. A revised version of the **By** parameter which gives more control over the iterations executed.
2. A truth-valued expression that when applied to a tuple in an iteration result, determines whether that tuple is to be used for another iteration or not.

10.1 A Revised ‘By’ Parameter

The **By** parameter proposed earlier enables the number of iterations to be specified when **Compose** is invoked. This can be particularly useful for operands representing tree and DAG structures as the number specified relates directly to the number of levels in the tree/DAG from the starting point.

However it would be useful if this approach were generalised so that the results of *required* iterations could be specified by this parameter. For example, if the user wants mappings of length 2, 4, and 6, then specifying 1, 3, and 5 iterations of **Compose** (in order to extend that many times the original mappings of length one) would provide the required result, as a distributed union of the results of the first, third, and fifth iterations. These iterations could correspond to levels in a tree/DAG.

To achieve this, let the **By** parameter be replaced by an **Iterate** parameter that takes a set of iterations as its value. So

\[
\text{Iterate} \{ 1, 3, 5 \}
\]

would specify the required iterations in the above example. Note that the parameter value would be a *set* of numbers even though it would typically more convenient to write it out as a *list* of numbers.

In principle **Compose** may have to generate the results of omitted iterations in order to derive the results of following iterations. So in the above example, the second and fourth iterations would be carried out to obtain the values of the third and fifth iterations, even though the results of the second and fourth iterations would not appear in the result.

If transitive closure arises before the full set of specified iterations is accomplished, then the complete result will be the distributed union of only those iteration results up to and including the final iteration that delivered a non-empty result. The following

---

\(^5\) In fact the departure time may need to be \( t \) hours after the arrival time, where \( t \) is the transfer time needed for passengers and luggage to transfer from one flight to the next.
iterations can be thought of as delivering empty relvalues as results, which therefore add nothing to the complete result.

The question then arises as to how to specify all the iterations up to and including transitive closure, or the last set of iterations leading up to and including transitive closure. The proposed answer is:

- Let there be a notation to specify sets of numbers, e.g. ‘m – n’ would specify the set of iterations m to n inclusive. m must be less than or equal to n.
- Let ‘+’ be usable as the number of the upper limit to specify that it is the iteration which delivers the final set of mappings that provide transitive closure.

This allows an Iterate parameter such as the following

Iterate[ 2 - 4, 6 - + ]

which would specify iterations 2 to 4 inclusive, and iterations 6 to the final transitive closure iteration inclusive.

For consistency, an Iterate parameter

Iterate[ + ]

would specify that the complete result consisted of only the final iteration.

10.2 ‘End’ Conditions

Looking at descriptions of Compose-type operators proposed in the literature [5, 9, 10], it is clear that Compose can be regarded as an operation carried out in a DO UNTIL or DO WHILE loop or as a recursion. In either case, successive compositions are executed until terminated by an end condition. It therefore seems sensible to add such an end condition as an optional parameter to the operator.

Examples of its use are:

- During the traversal of a route, a value might be calculated, and the end condition terminates compositions when that value exceeds (say) a certain amount; this might be when the total distance travelled from node A equals or exceeds 100 miles.

- The end condition terminates compositions when an attribute value found in a tuple has a particular value; this might be when node B (say) is reached.

The end condition should be any expression that evaluates to the Truth-type value. It should use the same notation as the With parameter, not only for consistency but also to allow it to use computed attribute values as well as attribute values taken from tuples.

It needs to be decided whether the end condition should be a WHILE or UNTIL condition, and whether it should be executed before or after each iteration. In making these decisions, it is worth bearing the following in mind:

- It is desirable to be able to prevent even the first iteration from contributing to the result (i.e. the complete result would be an empty relvalue) rather than always having to accept at least one iteration.

---

6 An end condition cannot be duplicated by applying a Restrict operator to the result of Compose, because the end condition is applied at every iteration of the composition, not after compositions are complete, which is the only place a Restrict operation could apply.
- It is preferable to use computed and tuple attribute values from the current iteration rather than the previous iteration, as this is simpler to bear in mind when constructing the condition.

It is therefore proposed that the end condition be a **While** condition, because this can be applied to the current iteration, and prevent the current iteration’s result from contributing to the complete result. Thus if

```
While[ End-Condition ]
```

returns **True**, then the result of the current iteration is included in the complete result, and another iteration can take place. If it returns **False**, then the result of the current iteration is omitted and no more iterations take place. Thus the **While** parameter for the above 2 examples would be:

```
Rel Compose[ Node1 ; Node2 ] With[ Distance <= [Distance + ]Distance ]
While[ ]Distance < 100 ]
```

and

```
Rel Compose[ Node1 ; Node2 ] With[ Distance <= [Distance + ]Distance ]
While[ ]Node2 ~= ‘B’ ]
```

making appropriate naming assumptions, and given that it will be the ‘Node2’ attribute of the right-hand operator that determines whether node ‘B’ has been reached or not. Of course both conditions could be applied together, say with an **Or**:

```
Rel Compose[ Node1 ; Node2 ] With[ Distance <= [Distance + ]Distance ]
While[ ]Distance < 100 Or ]Node2 ~= ‘B’ ]
```

Note that the **While** parameter is applied to each result tuple generated in an iteration, to determine whether it goes into the result or not. Therefore while it may terminate composition with respect to one result tuple generated, there may be other tuples for which the end condition is still **True** and which therefore enter the result and may give rise to another iteration\(^7\). Execution ceases altogether when no more result tuples are generated.

### 11. Starting Conditions

It has so far been assumed that the operand of **Compose** is a relvalue that is to be composed with itself. However an equally likely usage is that a relvalue representing the mappings of just a few graph structures - perhaps just one structure – is to be composed with a relvalue representing many structures. There are 2 possibilities:

```
Few Compose Many
```

and

```
Many Compose Few
```

where **Few** has a relvalue representing a small number of graphical structures and **Many** has a relvalue representing a large number. As an illustration, the former possibility arises if **Many** represents a BOM of all the company’s products and **Few** represents a small number of products that the user wishes to investigate. The latter possibility arises if **Many** represents the same BOM and **Few** represents the BOM of small number of unusual assemblies that are produced from (say) special

---

7 This action is desirable because if the composition is being used to follow multiple paths in a reconvergent graph, not all of these paths may require to be terminated at the same time.
manufacturing processes, and the user wishes to investigate which products involve these assemblies.

Such a dyadic operation requires the two operands to be type compatible, in order to permit the result relvalue from an iteration to be composed with the right-hand operand, and to permit iteration results to be unioned together to form the complete result. Typically the Few operand is likely to be a restricted version of the Many operand.

Thus it is proposed to make Compose a dyadic operator such that if no second operand is provided, it defaults to a pseudo monadic operator that uses 2 copies of the one operand.

Note that by definition, the result of a dyadic composition is always longer mappings whose source node is a source node in the left-hand operand.

Thus if the KeyBoth option is used (so as to discard result tuples whose key values duplicate those in an operand) then in the dyadic case it is the left-hand operand that is referred to, not the right-hand operand.

12. Other Comparisons

Instead of constraining comparisons to be ‘=’ comparisons, can the use of comparisons be generalised to include other types of comparison?

The traditional Compose operator considered so far can be considered to be based on the Natural Join operator. As shown earlier

\[ R \text{ Compose}(x; y) \]

could be considered to be executed by the expression

\[ R \text{ Join}(y \mid (R \text{ Rename}(y <- x; z <- y) \text{ Project}(\sim y) \text{ Rename}(y <- z))) \]

In RAQUEL there is also a Generalised (or Theta) Join operator, whose purpose is to provide joins based on any type of comparison and to permit calculations of attribute values to be used in the comparisons. So a reasonable starting point would be to consider whether the above defining expression of Compose could be amended to use a Generalised Join instead of a Natural Join.

To investigate this, consider first the flight example posed earlier, where an airline timetable is used to look for specific flights between Britain and New Zealand. Let a flight from Newcastle (UK) to Christchurch (New Zealand) be sought. Let there be a relvar named Timetable which contains the relevant data:

\[ \text{Timetable} (\text{DepAir}, \text{DepTime}, \text{ArrAir}, \text{ArrTime}, \text{Airline}) \]

with attributes:

- **DepAir** = Departure Airport
- **DepTime** = Departure Time
- **ArrAir** = Departure Airport
- **ArrTime** = Departure Time
- **Airline** = Name of the airline operating the flight

Let the type for airports be the set of international ICAO airport codes, the type for times be **DateTime** with the time component using UTC international standard time,

---

8 Traditional mathematical relations do not have to be of the same type, as long as the attributes being compared in each composition are of the same type.
and the type for airline names be \textbf{Text}. The sole key will consist of all the attributes except \textquote{Airline}.

If

\[
\text{Join(}\ y \text{)}
\]

is replaced by

\[
\text{Gen(}\ \text{expression} \text{)}
\]

in the defining expression for \textbf{Compose}, then the parameter expression of \textbf{Gen} to be used in the example would be

\[
\text{Gen(}\ \text{lDepAir} = \text{lArrAir And}\ \\
\text{lArrTime + Time[}2:00\text{]} \leq \text{lDepTime And}\ \\
\text{lArrTime + Time[}5:00\text{]} \geq \text{lDepTime} \text{)}
\]

The comparison of arrival and departure times is to achieve between 2 and 5 hours to transfer from one flight to the next\(^9\).

For convenience, the \textbf{With} parameter convention for the attribute names of left and right operands is used. This ostensibly replaces the need for the use of the \textbf{Rename} operator for the right-hand operator. However in terms of the underlying defining expression, \textbf{Rename} would be used in the same way for this version of \textbf{Compose} as it would be in the standard version of \textbf{Compose}.

The \textbf{Project} operator would likewise be used in the corresponding way to remove from the result the attributes corresponding to \textquote{LArrAir}, \textquote{LArrTime}, \textquote{LDepAir}, and \textquote{LDepTime}. The application of the second \textbf{Rename} operator to the result would deal with the consequences of the first \textbf{Rename} operation on the right-hand operand.

It can be seen that the attributes involved in the mapping are all those named in the \textbf{Gen} parameter expression. So if the contents of the \textbf{Gen} parameter expression were to be used as the main \textbf{Compose} parameter instead of the 2 sets of named attributes, then the \textbf{Compose} operator could function in the more general manner proposed.

To complete the example, the \textbf{Compose} operation could be written as follows:

\[
\text{Timetable } \textbf{Restrict(}\ \text{lDepAir = \textquote{EGNT} And}\ \\
\text{lDepTime} \geq \text{DateTime[}10 \text{ June } 2012; 10:00\text{]} \text{)}\ \\
\textbf{Compose(}\ \text{lDepAir = \text{lArrAir And}\ \\
\text{lArrTime + Time[}2:00\text{]} \leq \text{lDepTime And}\ \\
\text{lArrTime + Time[}5:00\text{]} \geq \text{lDepTime} \text{)}\ \\
\textbf{Iterate(}\ 1, 2, 3, 4\ \\
\textbf{While(}\ \text{lArrAir} \neq \textquote{NZCH}\ \\
\text{Timetable}\text{)}
\]

\textbf{Compose} is used dyadically, with its left-hand operand being a restricted version of \textbf{Timetable} that contains only those departing flights from Newcastle that leave after the specified time. A \textbf{While} parameter is included so that composition ceases after those flights that arrive at Christchurch airport. However no knowledge is assumed of the nature of the mappings held in \textbf{Timetable}, and it seems reasonable to suppose there may be loops in it. In this case the \textbf{While} parameter will not terminate composition, and so the \textbf{Iterate} parameter is included to prevent excessive results, the number of iterations being determined by the maximum number of flights that seem reasonable to reach Christchurch. No extended keys are desirable so the \textbf{Key} parameter is not used.

\(^9\) It is assumed the addition of hours to \textbf{DateTime} values can cope with any changes to the date which this may involve.
The result consists of mappings representing travel from Newcastle using 2 flights, 3 flights, 4 flights, and 5 flights. Each tuple will contain the non-mapping attribute ‘Airline’ whose value will be that of the name of the airline carrying out the last flight in the journey.

Further investigation may be needed to obtain the optimum flight information, but it is unreasonable to expect that from a standard relational operator. At least the operator makes such investigation feasible.

13. Examples of Use
Let us assume the following relvar :-

\[ Employees (\text{Empno, Ename, Sal, Mnger}) \]

where :

\[
\begin{align*}
\text{Empno} & = \text{Employee Identification Number (which hence is key).} \\
\text{Ename} & = \text{Employee Name.} \\
\text{Sal} & = \text{Salary of an employee.} \\
\text{Mnger} & = \text{Identification Number of the employee’s manager (which has the same data type as ‘Empno’).}
\end{align*}
\]

Let \( Employees \) hold data about employees, including a hierarchy of managerial relationships.

Each tuple in it represents a mapping between an employee and their manager. ‘Ename’ and ‘Sal’ are non-mapping attributes describing an employee (not a manager). The (only) key represents only one node of the mapping, not both.

Consider how \textit{Compose} could be used to create results containing the answers to the following queries on \textit{Employees} :-

1. Find each employee’s manager’s manager.

\[
\text{Employees \ Compose[ Empno; Mnger ] \ Project[ Empno, Mnger ]}
\]

(N.B. One composition occurs by default. The result provides the ID of the employee and their manager’s manager. To find each employee’s manager, \textit{Compose} is not needed; it is already in \textit{Employees}).

2. Get all the managers above employee ‘Smith’ in his chain of command.

\[
\begin{align*}
\text{Employees \ Restrict[ Ename = 'Smith'] } \\
\text{Compose[ Empno; Mnger ] \ Iterate[ 1 - + ] Employees Union ( Employees \ Restrict[ Ename = 'Smith'] )}
\end{align*}
\]

(N.B. The immediate manager of ‘Smith’ is already given in \textit{Employees} and so has to be added to those generated by \textit{Compose} with a \textit{Union}).

3. Get all the direct subordinates of employee ‘Smith’.

\[
\begin{align*}
\text{Employees \ Restrict[ Ename = 'Smith'] } \\
\text{Compose[ Mnger; Empno ] \ Employees}
\end{align*}
\]

(N.B. This query uses mappings in the reverse direction compared to the previous 2 queries. Mappings are from the node defined by the first set of attributes to the node defined by the second set of attributes. The result will contain the non-mapping
attribute values of subordinates, with their own IDs, and Smith’s ID as their ‘Mnger’ attribute value).

4. Get the total salary bill for all the subordinates, at whatever level, that ‘Smith’ is responsible for.

\[
\text{Employees } \text{Restrict[ Ename = ‘Smith’ ] } \\
\text{Compose[ Mnger; Empno ] } \text{Iterate[ 1 - + ] Employees } \\
\text{GroupBy[ With[ TotalSal } \leftarrow \text{ Bag[ Sal | Sum ] } \\
\text{(N.B. The Compose operation generates a relvalue containing a tuple for every employee below ‘Smith’ in the hierarchy, and each tuple contains the salary of that employee. Therefore a GroupBy^{10} operation is needed to sum all those salaries^{11}).}
\]

5. Find the IDs of all those employees with no subordinates (i.e. get all the leaf nodes).

\[
\text{Employees } \text{Project[ Empno ] } \\
\text{Diff } \\
\text{( Employees } \text{Compose[ Mnger; Empno ] ] } \\
\text{Project[ Mnger ] } \text{Rename[ Empno } \leftarrow \text{ Mnger ] ) } \\
\text{(N.B. The set of all employees with a subordinate (determined from one composition) is taken from the set of all employees. The attribute name change is required so that the operands of Diff have the same type).}
\]

6. Get all the managers above employee ‘Smith’ and their salaries, for all those with a salary of less than £50,000.

\[
\text{Employees } \text{Restrict[ Ename = ‘Smith’ ] } \\
\text{Compose[ Empno; Mnger ] } \text{Iterate[ 1..+ ] Employees } \\
\text{While[ Sal < 50000 ] Employees } \\
\text{Project[ Mnger, Ename, Sal ] } \\
\text{(N.B. The query relies on salaries increasing up the managerial hierarchy, so that a } \\
\text{composition can be terminated when a manager has a salary of £50,000 or more, } \\
\text{because that manager’s manager will have an even greater salary. The extra Union } \\
\text{carried out in query two is unnecessary here, because a manager’s salary is not in the employee’s tuple).}
\]

7. How far down (i.e. how many layers) is ‘Smith’ from the top of the hierarchy?

\[
\text{Emp } \leftarrow \text{View Employees } \text{Extend[ Level } \leftarrow \text{ 1 ] } \\
\text{Emp } \text{Restrict[ Ename = ‘Smith’ ] } \\
\text{Compose[ Empno; Mnger ] } \text{With[ Level } \leftarrow \text{ [Level } + \text{ ]Level ] } \\
\text{Iterate[ + ] Emp } \\
\text{Project[ Level ] }
\]

(N.B. A virtual relvar, Emp, whose value is that of Employees but with the additional ‘Level’ attribute always containing the number 1, is used for the composition.

---

^{10} Both Compose and GroupBy use a With parameter to calculate values.

^{11} By grouping the Compose result on an empty set of attributes, GroupBy treats the result as one group of tuples and returns a single tuple, containing just the attribute ‘TotalSal’, as its result. GroupBy treats the ‘Sal’ attribute as a Bag of values because some employees may have the same salaries as other employees, and no employee salary is to be omitted from the result.
\textbf{Compose} returns can then be counted up to find the number of layers. The count in the last iteration will contain the maximum number of layers traversed through).

It is worth noting that, because the relvar \textit{Employees} is designed to be about employees, the manager data is “part of the data about employees”. Consequently the mappings up the managerial hierarchy cannot be \textbf{Composed} in a way that is the symmetric reverse of mappings down the managerial hierarchy. Even though traversing up and down the hierarchy are logical opposites, the design of the relvar impedes this to a degree.

### 14. Syntax and Summary of ‘Compose’

The syntax of monadic \textbf{Compose} is:

\[
\text{RelValue} \ \text{Compose}[ \ \text{Mapping} \ ] \ \text{With}[ \ \text{Set-Attribute-Exp} \ ] \\
\text{KeyResult}[ \ \text{Name} \ \leftarrow \ \text{AttributeName-Set} \ ] \\
\text{Iterate}[ \ \text{Set} \ ] \ \text{While}[ \ \text{Logical-Exp} \ ]
\]

or

\[
\text{RelValue} \ \text{Compose}[ \ \text{Mapping} \ ] \ \text{With}[ \ \text{Set-Attribute-Exp} \ ] \\
\text{KeyBoth}[ \ \text{Name} \ \leftarrow \ \text{AttributeName-Set} \ ] \\
\text{Iterate}[ \ \text{Set} \ ] \ \text{While}[ \ \text{Logical-Exp} \ ] \ \text{RelValue}
\]

The syntax of dyadic \textbf{Compose} is:

\[
\text{RelValue} \ \text{Compose}[ \ \text{Mapping} \ ] \ \text{With}[ \ \text{Set-Attribute-Exp} \ ] \\
\text{KeyResult}[ \ \text{Name} \ \leftarrow \ \text{AttributeName-Set} \ ] \\
\text{Iterate}[ \ \text{Set} \ ] \ \text{While}[ \ \text{Logical-Exp} \ ] \ \text{RelValue}
\]

or

\[
\text{RelValue} \ \text{Compose}[ \ \text{Mapping} \ ] \ \text{With}[ \ \text{Set-Attribute-Exp} \ ] \\
\text{KeyBoth}[ \ \text{Name} \ \leftarrow \ \text{AttributeName-Set} \ ] \\
\text{Iterate}[ \ \text{Set} \ ] \ \text{While}[ \ \text{Logical-Exp} \ ]
\]

where:

- \textit{RelValue} is a relvar name, relational algebra expression, or a literal that evaluates to a relvalue.

  In the dyadic case, \textbf{Compose} maps from the left-hand operand to the right-hand operand in the first iteration, and thereafter from the previous iteration’s result to the right-hand operand. The 2 operands must have the same relational type.

  In the monadic case, \textbf{Compose} uses 2 copies of the left-hand operand. One copy is treated as the left-hand operand and the other as the right-hand operand; the 2 operands are then used as in the dyadic case.

- The \textit{[Mapping]} parameter can take 2 forms:
  - \texttt{[Node1; Node2]} where Node1 and Node2 are 2 sets of attribute names which represent the nodes between which mapping occurs, the direction of the mapping being from Node1 to Node2.
  - \texttt{[TruthValued-Exp]} where TruthValued-Exp is an algebraic expression whose values are those of named attributes appearing in the operands and literal values, and which must evaluate to a value of type \texttt{Truth}. Since named attributes represent the mapping, they must appear in the expression; the use of literal values is not mandatory but to enable the mapping to be properly defined.
The \textit{Mapping} parameter may have a \texttt{| prepended to it or a} \texttt{| appended to it (but not both). The additional \texttt{| indicates that any non-mapping attributes in the result are taken from the left-hand operand. The additional \texttt{| indicates that any non-mapping attributes in the result are taken from the right-hand operand. In either case, result attribute values may be overwritten by the results of assignments in the \texttt{With[ Set-Attribute-Exp]} parameter. If neither \texttt{| nor \texttt{|} is attached to the parameter, then \texttt{|} is assumed by default.

The \texttt{With[ Set-Attribute-Exp]} parameter is a set of assignments to attributes in the result. Each assignment is of an attribute expression that evaluates to an attribute value. Semi-colons are used to separate assignments.

The \texttt{KeyResult[ Name \leftarrow AttributeName-Set]} and \texttt{KeyBoth[ Name \leftarrow AttributeName-Set]} parameters specify a set of attributes that are to form the key of the result.

The \texttt{Iterate[ Set]} parameter contains one or more sets of positive integer numbers that define the iterations to be executed. If used, ‘+’ represents the integer value of the final iteration to produce a non-empty result in the generation of a transitive closure.

The \texttt{While[ Logical-Exp]} parameter is a truth-valued expression that must evaluate to \texttt{True} for another iteration to be carried out; if it evaluates to \texttt{False} no further iterations are executed.

The \texttt{Mapping} parameter is mandatory as \texttt{Compose} cannot function without this input.

The \texttt{With[ Set-Attribute-Exp]} parameter is optional. It is required if attribute values are to be calculated and put in the result.

The \texttt{KeyResult[ Name \leftarrow AttributeName-Set]}, \texttt{KeyBoth[ Name \leftarrow AttributeName-Set]}, \texttt{Iterate[ Set]}, and \texttt{While[ Logical-Exp]} are optional, terminating parameters. Iterations cease when one or more of them takes effect.

When \texttt{Compose} executes an iteration, each tuple in the left-hand operand is composed with as many tuples in the right-hand operand as logically possible. Any applicable terminating parameters may prevent such result tuples from forming part of the relvalue that is the iteration result. Iterations cease when the result of an iteration is an empty relvalue; an empty relvalue may arise due to transitive closure and/or the effects of a terminating parameter(s).

The result of a \texttt{Compose} operation is the distributed union of all the iteration results. It contains transitive mappings derived from the operands. Therefore to obtain a transitive relational result, the relevant operand must be unioned with the result of \texttt{Compose}.

15. Conclusions
It is useful to have some criteria by which to judge the above proposals for a \texttt{Compose} operator. Raquel uses the following 4 criteria with respect to concepts:
1. There should be the minimum number of concepts.
2. Each concept should be straightforward.
3. Each concept should be generalised as much as possible; or alternatively have no exceptions.
4. Each concept should be orthogonal to the other concepts so that it can be used in conjunction with any other concept.

Taking criterion 4 first, and equating ‘concept’ with ‘operator’, the criterion is satisfied in that Compose is just another operator, and can be combined with any other operator in a relational algebra expression. However the orthogonality aspect implies that Compose as an operator should execute a completely different concept to any other operator currently found in relational algebra. This aspect also is satisfied, especially if the concept is considered to be a series of compositions (‘iterations’) as opposed to just one composition.

Criterion 1 implies that Compose should reflect just one concept. If Compose merges 2 or more concepts, then they should be split apart and a different operator defined for each concept. However criterion 3 implies that Compose must be as general as possible. This suggests that the generalisation may involve combining related concepts, perhaps by seeing them as special cases of a single underlying concept. Applying criterion 2 appears to be the means of reconciling any clash between the implications of criteria 1 and 3.

The essential concept underlying Compose is that it treats 2 sets of attributes in its operands as representing nodes between which there is a direct non-transitive mapping. Its function is to generate longer, transitive mappings from the non-transitive mappings. We require all mappings of length two from the first iteration, all mappings of length three from the second iteration, and so on until we have the mappings of the required length, after which we want to stop.

Concerns arise when the mappings do not form a tree structure, in particular if they form a loop or a sequence of loops. We need to be able to control traverses round loops. Ideally the operator should check for the possibility of infinite loops, and terminate the operation if the terminating options provided to it do not prevent an infinite loop. Since Compose needs to be formally defined, termination needs to be based on a sound mathematical model.

To make Compose as general as possible, 4 factors need to be incorporated:

1. The starting position. This is determined by the operands.
2. The nature of the mappings. This is determined by the mandatory [Mapping] parameter. Compose supports both Natural and Generalised Join matching of nodes.
3. The calculation of attribute values during the course of mapping. This is optional, represented by the With[ Set-Attribute-Exp ] parameter, and may be omitted if not required.
4. The terminating conditions. They are optional, but stem from the wide range of possibilities, giving rise to 3 optional parameters.

A criticism of Compose as proposed is that it has one mandatory parameter plus up to 4 optional parameters and is therefore rather unwieldy and complex. Three of the 4 arise directly from the attempt to generalise the terminating conditions. It is these 3 that crucially create the problem of complexity.
In practice, a range of terminating conditions appears to be very important to cope with the potentially wide variety of graph structures to be traversed; if that variety is constrained then the operator loses some of its worth. So it would be worthwhile to consider whether a single parameter can subsume the 3 parameters. It is not immediately obvious how to do this.

One can also argue that:

- as only one parameter is mandatory, and typically not all the optional parameters will be used, in practice Compose is not so difficult to use as may be imagined;
- by having different parameters represent different conceptual options, a good balance between simplicity and general power is reached.

If Compose were to be simplified, the most desirable strategy to achieve it would be to break it down into 2 – 3 operators, which when combined together yielded the Compose function. For example, one approach might be to have a high-level ‘Iterator’ operator, which managed a sequence of operations including their termination, and which took a simplified ‘Compose’ operator as a parameter to carry out the composition in each iteration. However it is not yet clear how this might be achieved and how it would simplify usage, particularly as the replacement operators would need to provide useful functionality in their own right and be usable in combination with other currently-existing operators. If they could only be used together as a replacement for Compose, nothing would be gained.

Finally, a Compose operator of this nature provides a relational database with the Artificial Intelligence (AI) capability of Forward Chaining and Backward Chaining. The one operator can chain both forward and backward because it can be applied to mappings in either direction. The broad generality of Compose maximises its general AI usefulness.

**Appendix. Implementation of the ‘Compose’ Operator**

It is not the concern of this paper to consider implementation methods. A brief overview of them is given, taken from the literature, in order to show that it is practical to have such an operator.

The implementation methods that have been put forward may be classified as follows:

1. The use of bit-map matrices. This was originated by Warshall [1] and modified by Warren [2]. The method uses an \( n \times n \) matrix of elements \( a_{ij} \) where \( a_{ij} \) has a value of 1 if there exists a mapping from \( i \) to \( j \), and 0 otherwise. It computes the closure of the given graph by examining every element of the matrix exactly once, and if its value is 1 then making every successor of \( j \) a successor of \( i \).

2. Seminaive [3] and smart [4] algorithms. They work by repeatedly extending the known paths by adding new mappings to them via join operations. The smarter versions of the algorithms use more sophisticated techniques to minimise computation; they can also more directly take account of the precise nature of the Compose operation and maximise their efficiency with respect to it.
3. Graph-based algorithms. These use depth-first searches of a graph and produce topological sorts of the nodes. From Tarjan's original algorithm have been developed, for example, those of Ioannidis & Ramakrishnan [5] and Dar & Jagadish [8].

4. Path coding techniques. These generate codes for each node based on its ancestors using continued fractions [6] or hashing techniques [7] that relate the node to its ancestors.

See also Cruz & Norvell [9] for an overview of transitive closure algorithms from a different perspective.

Some algorithms would be better than others for executing any given query that a Compose operator might be used for. Factors that would affect the choice of algorithm include:

- The starting and finishing nodes, since either or both might permit significant portions of the graphical structure represented to be eliminated from the processing.
- The nature of the graphical structure represented, i.e. whether it was a tree structure, DAG, or DCG.
- The nature of the query, i.e. which of the following types of problem it represented:
  - reachability,
  - extremal path,
  - enumeration of paths and properties,
  - aggregation of paths and properties.
- The performance characteristics of the different algorithms and how well they lend themselves to the query at hand.

So ideally an implementation would have several algorithms available, just as there are typically several algorithms available to implement Join operations. Ideally too the algebraic optimiser would take into account the expression into which the Compose operation is embedded, in particular the use of any Restrict and GroupBy operations, in order to choose the most efficient implementation.

References

Investigating A Relational ‘Compose’ Operator  

David Livingstone


