# The Semantics of the Relational Algebra Assignments

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Principles</td>
<td>2</td>
</tr>
<tr>
<td>Introduction to the Semantics</td>
<td>3</td>
</tr>
<tr>
<td>Assignment of Properties</td>
<td></td>
</tr>
<tr>
<td>Binding Assignments</td>
<td></td>
</tr>
<tr>
<td>Storage Equation</td>
<td>6</td>
</tr>
<tr>
<td>View</td>
<td>8</td>
</tr>
<tr>
<td>Source Relvar</td>
<td>10</td>
</tr>
<tr>
<td>Sink Relvar</td>
<td>11</td>
</tr>
<tr>
<td>Physical Storage</td>
<td>13</td>
</tr>
<tr>
<td>Integrity Constraint Assignments</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>14</td>
</tr>
<tr>
<td>Key</td>
<td>16</td>
</tr>
<tr>
<td>Ad Hoc Constraint</td>
<td>19</td>
</tr>
<tr>
<td>Value Assignments</td>
<td></td>
</tr>
<tr>
<td>Real Relvar</td>
<td>22</td>
</tr>
<tr>
<td>Stored Relvar</td>
<td>24</td>
</tr>
<tr>
<td>Assignment to Manipulate Relvars</td>
<td></td>
</tr>
<tr>
<td>Value-Change Assignments</td>
<td></td>
</tr>
<tr>
<td>Relvalue</td>
<td>26</td>
</tr>
<tr>
<td>Amendment</td>
<td>27</td>
</tr>
<tr>
<td>Insertion</td>
<td>30</td>
</tr>
<tr>
<td>Deletion</td>
<td>32</td>
</tr>
<tr>
<td>Retrieving a Relvalue</td>
<td></td>
</tr>
<tr>
<td>Retrieval</td>
<td>34</td>
</tr>
<tr>
<td>Renaming a Relvalue</td>
<td></td>
</tr>
<tr>
<td>Rename</td>
<td>35</td>
</tr>
<tr>
<td>Removing a Relvar</td>
<td></td>
</tr>
<tr>
<td>Removal</td>
<td>37</td>
</tr>
<tr>
<td>Specifying a Relvar's Access Control</td>
<td></td>
</tr>
<tr>
<td>Access Control</td>
<td>39</td>
</tr>
</tbody>
</table>
General Principles
An assignment in RAQUEL carries out an action on its left-hand operand.

Typically the left-hand operand is the name of a relvar that is to receive the assignment. However RAQUEL allows the left-hand operand to be generalised as far as the semantics of the assignment permit - some assignments are only meaningful if the left-hand operand is a relvar name.

The generalisation takes 2 forms:

1. An Unnamed View.
   The left-hand operand is allowed to be a relational operator expression. The operand is treated as the definition of a view, and is called an unnamed view. (In The Third Manifesto, an unnamed view is termed a pseudovariable). All the real relvars that underlie it have the appropriate assignment action applied to them via the view updating mechanism.
   Unnamed views as left-hand operands are applicable to the <-- , <--Insert, <--Delete, and <--Amend assignments.

2. A Set of Relvars.
   The left-hand operand is allowed to be a set of relvars, in effect a set of left-hand operands. To avoid confusion, the set must be defined by one expression that forms the left-hand operand. The same assignment is made to all the relvars in the set in parallel, and only succeeds if it is successful for every relvar in the set.
   Sets of relvars as left-hand operands are applicable to all assignments, because the generalisation is unaffected by the meaning of the assignment.

Both forms of generalisation can arise in a left-hand operand at the same time; i.e. an unnamed view could represent a set of relvars, and each relvar in a set could be an unnamed view.

For example

\[ R \ \text{Project}[ RA ] \]

is a relational operator expression that projects the single attribute ‘RA’ from relvar \( R \), where ‘RA’ is a relation-valued attribute. If this expression formed the left-hand operand of an assignment, it would be an unnamed view that evaluated to a set of relvalues, one in each tuple of the projection.

Another example is

\[ \{ \{ Rexp1 \} \{ Rexp2 \} \ldots \{ Rexpn \} \} \]

which is a literal relvalue each of whose tuples is the relvalue of a relational operator expression, i.e. an unnamed view. A single named relvar is a special case of an unnamed view. This too could form the left-hand operand of an assignment.

Assignments to a set of left-hand operands can be expressed as follows:
Semantics of Relational Algebra Assignments

David Livingstone

20th November 2014 (9th March 2010)

{o \{n operands\} <--Assign

is a monadic assignment where

 <--Assign represents any standard or pseudo monadic assignment.

{o \{n operands\} <--Assign one-operand

is a dyadic assignment

where <--Assign represents any standard or pseudo dyadic assignment.

It might be thought reasonable to also permit

{o \{n operands\} <--Assign \{n operands\}

where n has the same value on both the left and right-hand sides. However sets impose no order or structure on their members. The syntactic order in which the members of the left and right-hand operand sets are written therefore has no semantic significance and cannot be used to associate a right-hand operand with its left-hand counterpart. Hence this form of assignment is not allowed1.

If the operands had been sequences

{<n operands>} <--Assign {<n operands>}

where {<…>} represents a sequence of items, then the assignment could have been carried out by associating the first right-hand operand with the first left-hand operand, and so on.

Clearly

one-operand <--Assign \{n operands\}

is illogical and so not permitted.

Introduction to the Semantics

For each assignment, this document gives a brief informal description of its semantics together with a formal specification that is expressed in terms of a set of pre conditions (i.e. what must be true beforehand in order to carry out a valid execution of the assignment) and a set of post conditions (i.e. what must be true after a valid execution of the assignment has been completed).

For convenience, this document describes the semantics of each assignment on the assumption that the left-hand operand is a single relvar. Where an unnamed view appears instead, it is assumed that the view update mechanism applies the assignment appropriately to all the underlying relvars. Where there is a set of left-hand operands, the assignment is applied to all those in the set.

The right-hand operand of dyadic assignments is always a relvalue – expressed as a literal relvalue, a relvar, or a relational algebra expression2.

1 If one could use the syntactic order for association, then left and right-hand operands could be associated.

2 Strictly speaking, a single literal relvalue and a single relvar are special cases of a relational algebra expression.
The action carried out by an assignment must provide a change of state to the left-hand operand (where the state change may be to the operand’s relational value, relational type, set of relational keys, set of relational ad hoc constraints, or relational binding).

Some assignments take parameters as well as operands. The requirements for the type and value of parameters are included in the pre conditions.

The document “The Semantics of Relations” gives the formal specification of a relvalue. As described there, a relvalue is described by the mapping

$$RV \mapsto \{RT, \{RK_k\}, \{RAC_c\}\}$$

where:
- $RV$ is its composite value,
- $RT$ is its reltype,
- $\{RK_k\}$ is its set of relational keys,
- $\{RAC_c\}$ is its set of ad hoc constraints.

In addition to its logical relational model, RAQUEL includes a physical storage model. The latter includes the Storage Mechanism that is used to bind a logical relvalue to its physically stored representation. As assignments are used to specify Storage Mechanisms, it is convenient to add Storage Mechanisms to the formal relational specification. Thus the above mapping is expanded to

$$RV \mapsto \{RT, \{RK_k\}, \{RAC_c\}, SM\}$$

where $SM$ is the Storage Mechanism with which $RV$ is bound.

It is convenient to treat operands as relvars, with the following consequent mappings to their composite values, integrity constraints, and bindings:

$$VAR \mapsto \{RV \mapsto \{RT, \{RK_k\}, \{RAC_c\}, SM\}\}$$

where $VAR$ is the relvar name.

Let the left-hand operand (which all assignments must have) be named $LOP$. It is necessary to distinguish the state of the left-hand operand after the assignment from its state before the assignment. Let the state before be designated $LOP$ and the state after be designated $LOP'$. Let the right-hand operand (which only dyadic assignments will have) be named $ROP$. Therefore the operands are represented by the following mappings:

- $LOP$ \mapsto \{LOV \mapsto \{LOT, \{LOK_k\}, \{LOAC_c\}, LOSM\}\}
- $LOP'$ \mapsto \{LOV' \mapsto \{LOT', \{LOK'_k\}, \{LOAC'_c\}, LOSM'\}\}
- $ROP$ \mapsto \{ROV \mapsto \{ROT, \{ROK_k\}, \{ROAC_c\}, ROSM\}\}

Thus the operands of the different kinds of assignment are as follows:
- $LOP'$ for standard monadic;
- $LOP'$ and $LOP$ for pseudo monadic;
- $LOP'$ and $ROP$ for standard dyadic;
- $LOP'$, $LOP$ and $ROP$ for pseudo dyadic.

---

3 This follows the convention used in Z Schema Calculus.
The LOP operand is not used by standard assignments because \( LOP' \) is not dependent on the prior state of the left-hand operand (i.e. \( LOP \)). Only pseudo assignments use \( LOP \) because then \( LOP' \) is dependent on the prior state of the left-hand operand (i.e. \( LOP \)), and \( ROP \) in the dyadic case.

Note that an assignment may appear to be a standard assignment as far as the prior value of its left-hand operand is concerned, but because it takes account of some other part of the prior state of the operand (e.g. its type), the assignment is in fact a pseudo assignment.

The difference between an assignment to a set of left-hand operands and an assignment to one left-hand operand can be formally specified:

- The following formally specifies an assignment to a set of operands:
  \[
  \begin{align*}
  & ( ( LOP' \text{Meta}[\text{Degree}] = 1 ) \land \\
  & ( LOP' \text{Meta}[\text{AttributeCat}] = \{ 'Relation' \} ) \\
  \Rightarrow \\
  & ( \{ \forall lop_i : LOP' \cdot lop_i <\text{-Assign} \} \lor \\
  & \{ \forall lop_i : LOP' \cdot lop_i <\text{-Assign ROP} \} )
  \end{align*}
  \]

- The following formally specifies the assignment to a single operand:
  \[
  \begin{align*}
  & ( ( LOP' \text{Meta}[\text{Degree}] \neq 1 ) \lor \\
  & ( 'Relation' \notin ( LOP' \text{Meta}[\text{AttributeCat}] ) ) \\
  \Rightarrow \\
  & ( LOP' <\text{-Assign} ) \lor ( LOP' <\text{-Assign ROP} )
  \end{align*}
  \]

In both cases, a pre condition implies a post condition.

No formal specification of the assignment to unnamed views is provided as this is a major topic in its own right.

The formal specifications of the following assignments specify the valid actions of the assignments. They exclude all consideration of what should happen if any errors arise.
The ‘Storage Equation’ Assignment

\[ LOP' \xrightarrow{=} \text{Equate} \ ROP \]

Informal Description
The purpose of the \( \xrightarrow{=} \text{Equate} \) assignment is to specify the physical binding of a real relvar, \( LOP' \), to the stored relvar(s) that hold(s) its value, the binding having the form of the relational operator expression \( ROP \). Thus to obtain the value of \( LOP' \), the expression \( ROP \) is executed.

The integrity constraints derivable from the expression \( ROP \) are also included in \( LOP' \).

The assignment corresponds to a view assignment, but is used to specify how the value of a real relvar is physically stored in fragments and/or merges of stored relvars.

Pre Conditions

\[ \text{o let } \{ \text{LeftExp} \} = \{ \text{token}_i : \text{LOP' Tokenise} \} \cdot \]
\[ \{ \text{LeftExp} \} \text{ Meta[ Cardinality ]} = 1 \land \]
\[ LOP' \text{ Meta[ Type ]} = \text{RelvarName} \land \]
\[ ( LOP \text{ Meta[ RelvarCat ]} = \text{Real} \lor \]
\[ LOP \text{ Meta[ Exists ]} = \text{False} ) \]

\( LOP' \) consists of one relvar name, which is either that of a pre-existing real relvar (as opposed to a stored, virtual, sink, or source relvar) or of a new relvar to be created.

\[ \text{o ( LOP \text{ Meta[ Exists ]} = \text{True } ) } \Rightarrow ( \text{LOT} = \text{ROT} ) \]

If \( LOP \) already exists, then \( ROP \) must have the same type (i.e. set of attribute names and types) as \( LOP \).

\[ \text{o let } \{ \text{RightExp} \} = \]
\[ \{ \text{token}_i : \text{ROP Tokenise} | \text{token}_i \text{ Meta[ Type ]} = \text{RelvarName} \} \cdot \]
\[ ( \forall \text{token}_i : \text{RightExp} \cdot \text{token}_i \text{ Meta[ RelvarCat ]} = \text{Stored} ) \land \]
\[ ( \{ \text{RightExp} \} \text{ Meta[ Cardinality ]} \geq 1 ) \]

\( ROP \) consists of an expression such that all relvar names in it are those of stored relvars, and there must be at least one such relvar in the expression.

Post Conditions

\[ \text{o ( LOP \text{ Meta[ Exists ]} = \text{False } \land } \]
\[ LOP' \Rightarrow \{ \text{LOV'} \Rightarrow \{ \text{LOT'}, \{ \text{LOK'}_k \}, \{ \text{LOAC'}_c \}, \text{LOSM'} \} \} \]
\[ \land ( \text{LOV'} = \text{ROV} ) \land \]
\[ ( \text{LOT'} = \text{ROT} ) \land \]
\[ ( \{ \text{LOK'}_k \} = \{ \text{ROK}_k \} ) \land \]
\[ ( \{ \text{LOAC'}_c \} = \{ \text{ROAC}_c \} ) \land \]
\[ ( \text{LOSM'} = \text{ROP Tokenise Parse} ) \land \]
\[ (LOP' \text{ Meta}[\text{RelvarCat}]) = \text{Real} \]

\[ \lor \]

\[ (LOP \text{ Meta}[\text{Exists}]) = \text{True} \land \]

\[ LOP' \rightarrow \{LOV' \rightarrow \{LOT'; \{LOK'_k\}, \{LOAC'_c\}, LOSM'\}\} \]

\[ | (LOV' = ROV) \land \]

\[ (LOT' = ROT) \land \]

\[ (\{LOK'_k\} = (\{LOK_k\} \cup \{ROK_k\})) \land \]

\[ (\{LOAC'_c\} = (\{LOAC_c\} \cup \{ROAC'_c\})) \land \]

\[ (LOP' \text{ Meta}[\text{RelvarCat}]) = \text{Real} \land \]

\[ (LOSM' = LOSM) \]

If \( LOP \) does not already exist, then \( LOP' \) is created with the value and type of \( ROP \), those keys and \( Ad \) \( Hoc \) constraints that are derivable from \( ROP \), a logical binding which consists of the parse tree of the expression \( ROP \), and is of category real.

If \( LOP \) does already exist, then \( LOP' \) takes the relvalue of \( ROP \). It retains its reltype (which is the same as that of \( ROP \)). Its keys and \( Ad \) \( Hoc \) constraints are retained, but any extra keys and \( Ad \) \( Hoc \) constraints derivable from \( ROP \) are added. It continues to be of category real and with the same logical binding as before.
The ‘View’ Assignment

\[ LOP' \leftarrow \text{View} \ ROP \]

Informal Description
The purpose of the \[ LOP' \leftarrow \text{View} \ ROP \] assignment is to specify the logical binding of a virtual relvar, \( LOP' \), to the real relvar(s) that hold(s) its value, the binding having the form of the relational operator expression \( ROP \). Thus to obtain the value of \( LOP' \), the expression \( ROP \) is executed.

The integrity constraints derivable from the expression \( ROP \) are also included in \( LOP' \).

The assignment corresponds to a storage equation assignment, but is used to specify how a virtual relvar can be defined in terms of one or more real relvars.

Named real relvars and named virtual relvars may both have their bindings changed over time. Since real relvars are the basis of a relational DB, such changes may include the alteration of the integrity constraints of a named virtual relvar, but this is not permissible for a named real relvar.

Pre Conditions

- Let \( \{ \text{LeftExp} \} = \{ \text{token}_i : LOP' \text{Tokenise} \} \cdot \)
  \( \{ \text{LeftExp} \} \text{Meta}[\text{Cardinality}] = 1 \land LOP' \text{Meta}[\text{Type}] = \text{RelvarName} \land ( LOP \text{Meta}[\text{RelvarCat}] = \text{Virtual} \lor LOP \text{Meta}[\text{Exists}] = \text{False} ) \)
  \( LOP' \) consists of one relvar name, which is either that of a pre-existing virtual relvar (as opposed to a real, stored, sink, or source relvar) or of a new relvar to be created.

- Let \( \{ \text{RightExp} \} = \{ \text{token}_i : ROP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \} \cdot \)
  \( ( \forall \text{token}_i : \text{RightExp} \cdot ( \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Real} ) \lor ( \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Virtual} ) ) \)
  \( \land ( \{ \text{RightExp} \} \text{Meta}[\text{Cardinality}] \geq 1 ) \)
  \( ROP \) consists of an expression such that all relvar names in it are those of real or virtual relvars, and there must be at least one such relvar in the expression.

Post Conditions

- \( ( LOP \text{Meta}[\text{Exists}] = \text{False} \land LOP' ) \rightarrow \{ \text{LOV} \} \rightarrow \{ \text{LOT'}, \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM'} \} \)
( LOV' = ROV ) \land
( LOT' = ROT ) \land
\{ \{ LOK_k \} = \{ ROK_k \} \} \land
\{ \{ LOAC_c \} = \{ ROAC_c \} \} \land
( LOSM' = ROP \text{ Tokenise Parse } ) \land
L{\text{OP'} Meta}[ \text{RelvarCat} ] = \text{Virtual})} \\
\lor
( L{\text{OP Meta}}[ \text{Exists} ] = \text{True} \land
L{\text{OP'}} \rightarrow \{ LOV' \rightarrow \{ LOT', \{ LOK_k \}, \{ LOAC_c \}, LOSM' \} \}
| ( LOV' = ROV ) \land
( LOT' = ROT ) \land
\{ \{ LOK_k \} = \{ ROK_k \} \} \land
\{ \{ LOAC_c \} = \{ ROAC_c \} \} \land
( LOSM' = ROP \text{ Tokenise Parse } ) \land
L{\text{OP'} Meta}[ \text{RelvarCat} ] = \text{Virtual})

If \text{LOP} does not already exist, then \text{LOP'} is created with the value and type of \text{ROP}, those keys and \text{Ad Hoc} constraints that are derivable from \text{ROP}, a logical binding which consists of the parse tree of the expression \text{ROP}, and is of category virtual.

If \text{LOP} does already exist, the same applies.
The ‘Source Relvar’ Assignment

\( LOP' \equiv \text{Source}[ \text{Physical-Location} ] \)

Informal Description
The purpose of the \( \equiv \text{Source} \) assignment is to specify that a relvar is a source and the physical location that it represents. The nature of ‘Physical-Location’ depends on the nature of the source. For example, a file or pipe may be represented by a Unix path.

‘Physical-Location’ is not yet fully defined. Currently it is taken to be text which can be evaluated to reference a physical location.

Pre Conditions

\[ \text{let } \{ \text{LeftExp} \} \equiv \{ \text{token}_i : LOP' \text{ Tokenise} \} \bullet \]
\[ \{ \text{LeftExp} \} \text{ Meta[ Cardinality ]} = 1 \land \]
\[ LOP' \text{ Meta[ Type ]} = \text{RelvarName} \land \]
\[ ( LOP \text{ Meta[ RelvarCat ]} = \text{Source} \lor \]
\[ LOP \text{ Meta[ Exists ]} = \text{False} ) \]

\( LOP' \) consists of one relvar name, which is either that of a pre-existing source relvar (as opposed to a real, virtual, sink, or stored relvar) or of a new relvar to be created.

Post Conditions

\[ ( \text{LOP Meta[ Exists ]} = \text{True} \land \]
\[ LOP' \rightarrow \{ \text{LOV}' \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \} \]
\[ ( \text{LOT}' = \text{LOT} \land \]
\[ ( \{ \text{LOK}'_k \} = \{ \text{LOK}_k \} \land \]
\[ ( \{ \text{LOAC}'_c \} = \{ \text{LOAC}_c \} ) \land \]
\[ ( \text{LOSM}' = ( \text{Physical-Location Tokenise Parse} ) ) \land \]
\[ ( \text{LOV}' = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) ) \land \]
\[ ( LOP' \text{ Meta[ RelvarCat ]} = LOP \text{ Meta[ RelvarCat ]} ) \]
\[ ) \]
\[ \lor \]
\[ ( \text{LOP Meta[ Exists ]} = \text{False} \land \]
\[ LOP' \rightarrow \{ \text{LOV}' \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}
\[ ( \{ \text{LOAC}'_c \} = \emptyset ) \land \]
\[ ( \text{LOSM}' = ( \text{Physical-Location Tokenise Parse} ) ) \land \]
\[ ( \text{LOV}' = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) ) \land \]
\[ ( \text{LOT} = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) \text{ Meta[ Attribute ]}) \land \]
\[ ( \{ \text{LOK}'_k \} = \]
\[ \text{Eval} ( \text{Physical-Location Tokenise Parse} ) \text{ Meta[ AttributeName ]} ) \]
\]
( \textit{LOP}' \texttt{Meta}[ \texttt{RelvarCat} ] = \texttt{Source} ) \)

If \textit{LOP} already exists, then \textit{LOP}' has the same type, the same set of keys, and the same set of \textit{Ad Hoc} constraints as \textit{LOP}, its binding is that specified by ‘Physical-Location’, the value of \textit{LOP}' is that physically stored at ‘Physical-Location’, and it retains its original category.

If \textit{LOP} does not already exist, then \textit{LOP}' has an empty set of \textit{Ad Hoc} constraints, the binding specified by ‘Physical-Location’, the value that is physically stored at ‘Physical-Location’, a type consisting of all the attribute names and types appearing in the value, a set of one key that comprises all the attributes in its type, and is of category source.

The ‘Sink Relvar’ Assignment

\textit{LOP}' \texttt{Sink}[ \texttt{Physical-Location} ]

\textbf{Informal Description}

The purpose of the \texttt{Sink} assignment is to \textit{specify that a relvar is a sink and the physical location that it represents}. The nature of ‘Physical-Location’ depends on the nature of the source. For example, a file or pipe may be represented by a Unix path.

‘Physical-Location’ is not yet fully defined. Currently it is taken to be text which can be evaluated to reference a physical location.
Pre Conditions

- Let \( \{ \text{LeftExp} \} == \{ \text{token}_t : LOP' \text{ Tokenise} \} \)
  - \( \{ \text{LeftExp} \} \text{ Meta[ Cardinality ] } = 1 \) \( \land \)
  - \( LOP' \text{ Meta[ Type ] } = \text{RelvarName} \) \( \land \)
  - \( ( LOP \text{ Meta[ RelvarCat ] } = \text{Sink} \) \( \lor \)
    - \( LOP \text{ Meta[ Exists ] } = \text{False} \)

  \( LOP' \) consists of one relvar name, which is either that of a pre-existing sink relvar (as opposed to a real, virtual, source, or stored relvar) or of a new relvar to be created.

Post Conditions

- \( ( LOP \text{ Meta[ Exists ] } = \text{True} \) \( \land \)
  - \( LOP' \xrightarrow{} \{ \text{LOV'} \xrightarrow{} \{ \text{LOT'}, \{ \text{LOK'}_k \}, \{ \text{LOAC'}_c \}, \text{LOSM'} \} \} \)
    - \( ( \text{LOT'} = \text{LOT} ) \) \( \land \)
    - \( ( \{ \text{LOK'}_k \} = \{ \text{LOK}_k \} ) \) \( \land \)
    - \( ( \{ \text{LOAC'}_c \} = \{ \text{LOAC}_c \} ) \) \( \land \)
    - \( ( \text{LOSM'} = ( \text{Physical-Location Tokenise Parse} ) ) \) \( \land \)
    - \( ( \text{LOV'} = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) ) \) \( \land \)
    - \( ( LOP' \text{ Meta[ RelvarCat ] } = LOP \text{ Meta[ RelvarCat ] } ) \)

- \( ( LOP \text{ Meta[ Exists ] } = \text{False} \) \( \land \)
  - \( LOP' \xrightarrow{} \{ \text{LOV'} \xrightarrow{} \{ \text{LOT'}, \{ \text{LOK'}_k \}, \{ \text{LOAC'}_c \}, \text{LOSM'} \} \)
    - \( ( \{ \text{LOAC'}_c \} = \emptyset ) \) \( \land \)
    - \( ( \text{LOSM'} = ( \text{Physical-Location Tokenise Parse} ) ) \) \( \land \)
    - \( ( \text{LOV'} = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) ) \) \( \land \)
    - \( ( \text{LOT'} = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) \text{ Meta[ Attribute ] } ) \) \( \land \)
    - \( ( \{ \text{LOK'}_k \} = \text{Eval} ( \text{Physical-Location Tokenise Parse} ) \text{ Meta[ AttributeName ] } ) \)
    - \( ( LOP' \text{ Meta[ RelvarCat ] } = \text{Sink} ) \)

If \( LOP \) already exists, then \( LOP' \) has the same type, the same set of keys, and the same set of Ad Hoc constraints as \( LOP \), its binding is that specified by ‘Physical-Location’, the value of \( LOP' \) is that physically stored at ‘Physical-Location’, and it retains its original category.

If \( LOP \) does not already exist, then \( LOP' \) has an empty set of Ad Hoc constraints, the binding specified by ‘Physical-Location’, the value that is physically stored at ‘Physical-Location’, a type consisting of all the attribute names and types appearing in the value, and a set of one key that comprises all the attributes in its type, and is of category sink.
The ‘Physical Storage’ Assignment

\[ LOP' == \text{Physical[ Storage-Mechanism ]} \]

Informal Description

The purpose of the \(==\text{Physical}\) assignment is to \textit{specify the physical storage mechanism} to be used to store the relvalue of \(LOP'\). Thus \(LOP'\) must be a stored relvar. ‘Storage-Mechanism’ consists of whatever set of parameters are needed to define a specific storage mechanism, and may include not only the particular nature of the mechanism – e.g. B-tree as opposed to hash file – but also any particular factors needed - e.g. file size and location.

‘Storage-Mechanism’ is not yet fully defined. Currently it is taken to be text which can be evaluated to reference a physical location.

Pre Conditions

- let \(\{ \text{LeftExp} \} == \{ \text{token}_t : LOP' \text{Tokenise} \} \)
  - \(\{ \text{LeftExp} \}\) Meta[ Cardinality ] = 1 \(\wedge\)
  - \(LOP'\) Meta[ Type ] = RelvarName \(\wedge\)
  - ( \(LOP \) Meta[ RelvarCat ] = Stored \(\lor\)
    - \(LOP \) Meta[ Exists ] = \text{False} )

  \(LOP'\) consists of one relvar name, which is either that of a pre-existing stored relvar (as opposed to a real, virtual, source, or sink relvar) or of a new relvar to be created.

Post Conditions

- ( \(LOP \) Meta[ Exists ] = \text{True} \(\wedge\)
  - \(LOP' \)\(\rightarrow\) \(\{ \text{LOV'} \} \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}
  - ( \(LOV'\) Meta[ Cardinality ] = 0 ) \(\wedge\)
    - ( \(LOT' = \text{LOT} \) ) \(\wedge\)
    - ( \(\{ \text{LOK}'_k \} = \{ \text{LOK}_k \} \) ) \(\wedge\)
    - ( \(\{ \text{LOAC}'_c \} = \{ \text{LOAC}_c \} \) ) \(\wedge\)
    - ( \(LOSM' = (\text{Storage-Mechanism Tokenise Parse}) \) ) \(\wedge\)
    - ( \(LOP'\) Meta[ RelvarCat ] = \(LOP\) Meta[ RelvarCat ] )
  ) \(\lor\)

- ( \(LOP \) Meta[ Exists ] = \text{False} \(\wedge\)
  - \(LOP' \)\(\rightarrow\) \(\{ \text{LOV'} \} \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}
  - ( \(LOV'\) Meta[ Cardinality ] = 0 ) \(\wedge\)
    - ( \(LOT' = \text{Undefined} \) ) \(\wedge\)
    - ( \(\{ \text{LOK}'_k \} = \text{Undefined} \) ) \(\wedge\)
    - ( \(\{ \text{LOAC}'_c \} = \emptyset \) ) \(\wedge\)
    - ( \(LOSM' = (\text{Storage-Mechanism Tokenise Parse}) \) ) \(\wedge\)
    - ( \(LOP'\) Meta[ RelvarCat ] = Sink )
  )
If $LOP$ already exists, then the value of $LOP'$ is an empty relvalue since the assignment of a storage mechanism means that there cannot yet be any data values associated with the relvar. $LOP'$ has the same type, the same set of keys, and the same set of Ad Hoc constraints as $LOP$, its binding is that specified by the assignment, and it retains its original category.

If $LOP$ does not already exist, then $LOP'$ has an empty relvalue since the assignment of a storage mechanism means that there cannot yet be any data values associated with the relvar. The type of $LOP'$ is undefined, and consequently its set of keys is undefined as even a default key cannot be defined. $LOP'$ has an empty set of Ad Hoc constraints, the binding specified by the assignment, and is of category sink.

The ‘Attribute’ Assignment

$$LOP' \leftarrow \text{Attribute}[ \{ an_j \leftarrow \text{Type}_j \} ]$$

Informal Description

The purpose of the $\leftarrow \text{Attribute}$ assignment is to specify the names and scalar data types of a relvar’s attributes. If an attribute has a relational type, then $\leftarrow \text{Attribute}$ is applied again inside the parameter; this can be continued recursively to any finite depth. The specification constitutes a relvar’s relational type (= reltype). $LOP'$ must be a relvar name.

As a relvar has a set of attributes, the order in which the attributes are specified is not significant (except where it is convenient to use the attribute order for some defaults).
If the parameter is empty, $LOP'$ is made attributeless.

If $LOP'$ does not already exist, the assignment creates it. If it does already exist, the previous set of attributes is replaced by the new set, which must be consistent with all of the relvar’s other pre-existing properties or the assignment fails with an error.

**Pre Conditions**

- let $\{ \text{LeftExp} \} = \{ \text{token}_i : LOP' \text{ Tokenise} \} \cdot$
  - $\{ \text{LeftExp} \} \text{Meta}[ \text{Cardinality}] = 1$ \wedge
  - $LOP' \text{Meta}[ \text{Type}] = \text{RelvarName}$
    - $LOP'$ consists of one relvar name.
  - $\{ \text{Type}_j \} \subseteq \text{DBMSInstallTypeSet}$
    - All scalar types specified for attributes are in the set provided by the DBMS installation (i.e. in DBMSInstallTypeSet).

**Post Conditions**

- $(LOP \text{Meta}[ \text{Exists}] = \text{False} \wedge$
  
  $LOP' \Rightarrow \{ \text{LOV}' \Rightarrow \{ \text{LOT}', \{ \text{LOK}'^k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}$
  
  $\mid (\text{LOV}' \text{Meta}[ \text{Cardinality}] = 0) \wedge$
  
  $(\text{LOT} =$
  
  $\{ \{ \text{an}_j : \text{AttributeName}, \text{Type}_j : \text{AttributeType} \}\}_j \} \wedge$
  
  $(\{ \text{LOK}'_k \} = \{ \{ \text{an}_j \}\}_j \}) \wedge$
  
  $(\{ \text{LOAC}'_c \} = \emptyset) \wedge$
  
  $(LOP' \text{Meta}[ \text{RelvarCat}] = \text{Real}) \wedge$
  
  $(\text{LOSM}' = \text{RealDefault}))$

  $\forall$

- $(LOP \text{Meta}[ \text{Exists}] = \text{True} \wedge$
  
  $LOP' \Rightarrow \{ \text{LOV}' \Rightarrow \{ \text{LOT}', \{ \text{LOK}'^k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}$
  
  $\mid (\text{LOV}' = \text{LOV}) \wedge$
  
  $(\text{LOT} =$
  
  $\{ \{ \text{an}_j : \text{AttributeName}, \text{Type}_j : \text{AttributeType} \}\}_j \} \wedge$
  
  $(\{ \{ \text{LOK}'_k \} = \{ \text{LOK}'_k \}\}) \wedge$
  
  $(\forall_k (\forall_j \text{a}'_j : \text{LOK}'_k \bullet \text{a}'_j \in \text{LOT}_j)_k) \wedge$
  
  $(\{ \{ \text{LOAC}'_c \} = \{ \text{LOAC}'_c \}\}) \wedge$
  
  $(\forall_k (\forall_j \text{a}'_j : \text{LOAC}'_c \bullet \text{a}'_j \in \text{LOT}_j)_k) \wedge$

  $(\text{LOSM}' = \text{LOSM}) \wedge$
  
  $(LOP' \text{Meta}[ \text{RelvarCat}] = LOP \text{Meta}[ \text{RelvarCat}])$

  $\}^{\wedge}$

  If $LOP$ does not already exist, then $LOP'$ is created, with a relvalue empty of tuples, the reltype
specified by the parameter of the assignment, a set of one key that comprises all the attributes in $LOP'$, and an empty set of Ad Hoc constraints. By default $LOP'$ is given the category real, and the binding specified as the default for real relvars.

If $LOP$ already exists, then $LOP'$ retains its original relvalue, but is given the reltype specified by the parameter of the assignment, so the two must be consistent. $LOP'$ retains the original set of keys (which must each use the same set of attributes, in name and type, used in $LOP$), the original set of Ad Hoc constraints (which must each use the same set of attributes, in name and type, used in $LOP$), the original binding, and the original relvar category.

The ‘Key’ Assignment

$LOP' \leftarrow \text{Key}[$ $\{ \text{KName}_{j1} \leftarrow \text{AttributeSet}_{j1} \}$ $; \sim \{ \text{KName}_{j2} \}$ $]$

Informal Description

The purpose of the $\leftarrow \text{Key}$ assignment is to specify and/or remove one or more relvar Keys. The parameter consists of a set of specifications of new keys and/or a set of key removals. The members of both sets can appear in any order in the parameter. $LOP'$ must be a relvar name.

Each key specification consists of the key’s name and the set of attributes comprising it; the order of the attribute names listed for a key is not significant since the key is a set of attributes.

---

$^4$ Every Key is a Candidate Key since RAQUEL possesses no other sort of key.
Semantics of Relational Algebra Assignments

The removal of a key(s) is specified by a set of one or more key names prefixed with ‘~’.

If a relvar has several keys, they can be assigned in separate <=Key assignments, all in one assignment, or in whatever sequence of assignments is preferred.

A key specification replaces an existing one of the same name.

If LOP’ does not already exist, the assignment creates it. If it does already exist, the resulting set of keys assigned must be consistent with the other properties of LOP or the assignment fails with an error.

Until a relvar has been assigned at least one key, by default all the attributes in it are taken to be the sole key; this default is dropped as soon as at least one key has been assigned.

Pre Conditions

1. Let \{LeftExp\} == \{token_{i} : LOP’ Tokenise \} •
   \{LeftExp\} Meta[ Cardinality ] = 1 ∧
   LOP’ Meta[ Type ] = RelvarName

   LOP’ consists of one relvar name.

2. ( ( LOP Meta[ Exists ] = False ) ∧
   \{ KName_{j2} \} = \emptyset ) ) ∨

   ( ( LOP Meta[ Exists ] = True ) ∧
   \{ LOK_{k} \} ⊇ \{ KName_{j2} \} ) ∧

   ( ( \{ KName_{j1} \} Intersect \{ KName_{j2} \} ) = \emptyset )

   If LOP does not already exist, then the keys to be removed must form an empty set.

   If LOP already exists, then any keys to be removed must already be in the set of keys possessed by LOP, and the set of keys to be created and the set to be removed must be disjoint.

Post Conditions

1. ( LOP Meta[ Exists ] = False ∧
   LOP’ \rightarrow \{ LOV’ \} → \{ LOT’, \{ LOK’_{k} \}, \{ LOAC’_{c} \}, LOSM’ \} )

   \{ LOV’ = Undefined \} ∧

   ( LOT’ =
    ( \{ AttributeSet_{j1} \} Dist[ Union ] ) → Unknown ) ∧

   ( \{ LOK’_{k} \} =
    \{ ∀_{k} kn_{k} : \{ KName_{j1} \} , as_{k} : \{ AttributeSet_{j1} \} •
    kn_{k} → as_{k} \} ) ) ∧

   ( \{ LOAC’_{c} \} = \emptyset ) ∧
( \text{LOP'} \text{Meta}[ \text{RelvarCat }] = \text{Real} ) \land \\
( \text{LOSM}' = \text{RealDefault} ) \\

\lor \\
( \text{LOP Meta}[ \text{Exists} ] = \text{True} \land \\
\text{LOP}' \rightarrow \{ \text{LOV}' \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \} \\
| ( \text{LOV}' = \text{LOV} ) \land \\
( \text{LOT}' = \text{LOT} ) \land \\
( \{ \text{LOK}'_k \} = \{ \text{LOK}_k \} ) \land \\
\{ \forall_k \text{ loc}_k : \{ \text{LOK}_k \} | \\
\text{dom}( \text{loc}_k ) \in ( \text{dom}( \{ \text{LOK}_k \} ) \setminus \{ \text{KName}_{j2} \}) \} \\
\text{Union} \\
\{ \forall_{j1} \text{ kn}_{j1} : \{ \text{KName}_{j1} \} , \text{as}_{j1} : \{ \text{AttributeSet}_{j1} \} \cdot \\
\text{kn}_{j1} \rightarrow \text{as}_{j1} \} \land \\
( \{ \text{LOAC}'_c \} = \{ \text{LOAC}_c \} ) \land \\
( \text{LOSM}' = \text{LOSM} ) \land \\
( \text{LOP'} \text{Meta}[ \text{RelvarCat }] = \text{LOP Meta}[ \text{RelvarCat} ] ) \\

\)

If \text{LOP} does not already exist, then \text{LOP}' is created with an undefined relvalue and a partially defined reltype (consisting of all the named attributes appearing in the specified keys, but with undefined types), and with the set of keys specified by the parameter of the assignment. \text{LOP}' is given an empty set of \textit{Ad Hoc} constraints, and by default the category real and the binding specified as the default for real relvars.

If \text{LOP} already exists, then \text{LOP}' retains its original relvalue and reltype, but its set of keys is revised according to what is specified by the parameter of the assignment. \text{LOP}' retains its original set of \textit{Ad Hoc} constraints, its original binding, and its original category.
The ‘Ad Hoc Constraint’ Assignment

\[ LOP' \Leftarrow\Leftarrow\text{Constraint[ CName ] } ROP \]

or

\[ LOP' \Leftarrow\Leftarrow\text{Constraint[ ~{ CName}_j ]} \]

Informal Description

The purpose of the dyadic \( \Leftarrow\Leftarrow\text{Constraint} \) assignment is to \textit{specify an Ad Hoc constraint for LOP’ and give it a name.}

The purpose of the monadic \( \Leftarrow\Leftarrow\text{Constraint} \) assignment is to \textit{remove a set of named Ad Hoc constraints}, namely all those given in the parameter.

In the dyadic case, \( LOP' \) is assigned the \textit{Ad Hoc} constraint expressed by \( ROP \) and given the identifier ‘CName’. \( ROP \) is a truth-valued relational operator expression. Whenever the relvalue of \( LOP' \) is altered, \( ROP \) is executed after the change of state. If it evaluates to \textit{True}, the change is valid and accepted; if it evaluates to \textit{False}, the change is invalid, rejected with an error, and the state of \( LOP' \) returned to the (original) state of \( LOP \).

‘#’ and ‘@’ may be incorporated into \( ROP \). Use of both ‘#’ and ‘@’ together in \( ROP \) enables the state change itself to be constrained.

If \( LOP \) does not already exist, the assignment creates it. If \( LOP \) does already exist, then the constraint, with its new name, is added to the set of \textit{Ad Hoc constraints} already possessed by \( LOP' \). If \( LOP \) already exists, the constraint is automatically applied to its current relvalue (unless the constraint includes ‘@’, because the previous value of \( RCOP \) is not available); if the constraint does not evaluate to \textit{True}, the assignment fails with an error.

Each constraint name must be unique to its recipient operand, but different relvars in a DB can use the same constraint names.

Referential integrity constraints are applied using \( \Leftarrow\Leftarrow\text{Constraint} \). This allows such constraints to be generalised to any kind of inclusion constraint.

Pre Conditions – Both Versions

- let \( \{ \text{LeftExp} \} = \{ \text{token}_t : LOP' \text{ Tokenise } \} \bullet \)

  \[ \{ \text{LeftExp} \} \text{ Meta[ Cardinality ]} = 1 \land LOP' \text{ Meta[ Type ]} = \text{RelvarName} \]

  \( LOP' \) consists of one relvar name.

Pre Conditions – Dyadic Version

- let \( \{ \text{RightExp} \} = \)

  \[ \{ \text{token}_t : ROP \text{ Tokenise } | \text{token}_t \text{ Meta[ Type ]} = \text{RelvarName} \} \bullet \]

  \( ( \forall \text{token}_t : \text{RightExp} \bullet ( \text{token}_t \text{ Meta[ RelvarCat ]} = \text{Real} ) \lor ( \text{token}_t \text{ Meta[ RelvarCat ]} = \text{Virtual} ) \lor ( \text{token}_t \text{ Meta[ RelvarCat ]} = \text{Sink} ) ) \)
ROP consists of an expression such that all relvar names in it are those of real, virtual or sink relvars. Note there is no requirement for any relvars to appear in the expression.

- \((\text{ROP Tokenise Parse}) \text{ Meta[ Degree ] } = \emptyset\)
  
  ROP comprises an expression whose type is a relvalue with no attributes, i.e. a relvalue that expresses a relational truth value.

- \(\text{CName} \notin \text{dom( } \{\text{LOAC}_c\} \}\)
  
  The new Ad Hoc constraint must not have a name that already identifies an Ad Hoc constraint possessed by LOP.

### Pre Conditions – Monadic Version

- \(\text{LOP Meta[ Exists ] } = \text{ True}\)
  
  The relvar LOP must already exist.

- \(\{ \text{CName}_i \} \subseteq \text{dom( } \{\text{LOAC}_c\} \}\)
  
  The set of Ad Hoc constraints in LOP must include all those named in the parameter of the assignment.

### Post Conditions – Dyadic Version

- \(\text{LOP Meta[ Exists ] } = \text{ False } \land \)
  
  \(LOP' \rightarrow \{ \text{LOV'} \rightarrow \{ \text{LOT'}, \{ \text{LOK'}_k \}, \{ \text{LOAC'}_c \}, \text{LOSM'} \} \} \)
  
  | ( \(\text{LOV' Meta[ Cardinality ] } = 0\) ) \land
  
  ( \(\text{LOT'} = \text{Undefined}\) ) \land
  
  ( \(\{ \text{LOK'}_k \} = \text{Undefined}\) ) \land
  
  ( \(\{ \text{LOAC'}_c \}\) =
    
    ( \(\{ \text{CName } \rightarrow (\text{ROP Tokenise Parse}) \}\) ) ) ) \land
  
  ( \(\text{LOP' Meta[ RelvarCat ] } = \text{Real}\) ) \land
  
  ( \(\text{LOSM'} = \text{RealDefault}\) )

  \)

- \(\lor\)

- \(\text{LOP Meta[ Exists ] } = \text{ True } \land \)
  
  \(LOP' \rightarrow \{ \text{LOV'} \rightarrow \{ \text{LOT'}, \{ \text{LOK'}_k \}, \{ \text{LOAC'}_c \}, \text{LOSM'} \} \} \)
  
  | ( \(\text{LOV'} = \text{LOV}\) ) \land
  
  ( \(\text{LOT'} = \text{LOT}\) ) \land
  
  ( \(\{ \text{LOK'}_k \} = \{ \text{LOK}_k \}\) ) \land
  
  ( \(\{ \text{LOAC'}_c \} = (\{ \text{LOAC}_c\} \text{ Union}
    
    ( \(\{ \text{CName } \rightarrow (\text{ROP Tokenise Parse}) \}\) ) ) ) ) \land
  
  ( \(\text{LOSM'} = \text{LOSM}\) ) \land
  
  ( \(\text{LOP' Meta[ RelvarCat ] } = \text{LOP Meta[ RelvarCat ]}\) )

  \)
If $LOP$ does not already exist, then $LOP'$ is created, with a relvalue empty of tuples, and an undefined reltype and an undefined set of keys since neither can be deduced from the assignment of an $Ad Hoc$ constraint. The $Ad Hoc$ constraint specified by the assignment is assigned to $LOP'$ as its only $Ad Hoc$ constraint. By default $LOP'$ is given the category real, and the binding specified as the default for real relvars.

If $LOP$ already exists, then $LOP'$ retains its original relvalue, its original reltype, and its original set of keys. The $Ad Hoc$ constraint specified by the assignment is added to its set of $Ad Hoc$ constraints. Its original binding and relvar category are retained.

Post Conditions – Monadic Version

- $(LOP \ Meta[ Exists ] = True \land
  LOP' \rightarrow \{ LOV' \rightarrow \{ LOT', \{ LOK'_k \}, \{ LOAC'_c \}, LOSM' \} \})$
  
  $\mid ( LOV' = LOV ) \land$
  
  $( LOT' = LOT ) \land$
  
  $(\{ LOK'_k \} = \{ LOK_k \}) \land$
  
  $(\{ LOAC'_c \} = (\{ LOAC_c \} \ Diff$
  
  \{ locm_j : \{ LOAC_c \} \mid \dom( locm_j ) \in \{ CName_j \} \}) ) \lor$
  
  $( LOSM' = LOSM ) \land$
  
  $(LOP' \ Meta[ RelvarCat ] = LOP \ Meta[ RelvarCat ] )$

If $LOP$ already exists, and those of its set of $Ad Hoc$ constraints identified by $\{ CName_j \}$ are removed in $LOP'$, and everything else is left unchanged in $LOP'$. 
The ‘Real Relvar’ Assignment

\[ LOP' \leftarrow \text{Real} \ ROP \]

Informal Description

The purpose of the \( \leftarrow \text{Real} \) assignment is to create a real (or base) relvar with the name of \( LOP' \) and the value of \( ROP \). Any integrity constraints derived from \( ROP \) are also included in \( LOP' \).

If a real relvar of the same name already exists, its value and corresponding integrity constraints are replaced by those derived from \( ROP \); any pre-existing integrity constraints that are not replaced remain, but the assigned relvalue must also satisfy them or an error results. An error also results if a relvar of the same name already exists and is not a real relvar.

Pre Conditions

\[\begin{align*}
\text{o } & \quad \text{let } \{ \text{LeftExp} \} == \{ \text{token}_i : LOP' \text{ Tokenise } \} \bullet \\
& \quad \{ \text{LeftExp} \} \text{ Meta[ Cardinality ]} = 1 \land \\
& \quad \text{LOP' Meta[ Type ]} = \text{RelvarName} \\
& \quad \text{LOP' } \text{ consists of one relvar name.}
\end{align*}\]

\[\begin{align*}
\text{o } & \quad ( \text{LOP Meta[ Exists ]} = \text{False} ) \\
\text{o } & \quad ( \text{LOP Meta[ Exists ]} = \text{True} \land \\
& \quad ( \text{LOP Meta[ RelvarCat ]} = \text{Real} ) ) \\
\end{align*}\]

Either \( LOP \) does not already exist, or it already exists as a real relvar.

\[\begin{align*}
\text{o } & \quad \text{let } \{ \text{RightExp} \} == \\
& \quad \{ \text{token}_j : ROP \text{ Tokenise } | \text{token}_j \text{ Meta[ Type ]} = \text{RelvarName } \} \bullet \\
& \quad ( \forall \text{token}_j : \text{RightExp} \bullet ( \text{token}_j \text{ Meta[ RelvarCat ]} = \text{Real} ) \\
& \quad \lor ( \text{token}_j \text{ Meta[ RelvarCat ]} = \text{Virtual} ) \\
& \quad \lor ( \text{token}_j \text{ Meta[ RelvarCat ]} = \text{Sink} ) ) \\
& \quad \text{ROP consists of an expression such that all relvar names in it are those of real, virtual or sink relvars. Note there is no requirement for any relvars to appear in the expression.}
\end{align*}\]

Post Conditions

\[\begin{align*}
\text{o } & \quad ( \text{LOP Meta[ Exists ]} = \text{False} \land \\
& \quad \text{LOP' } \rightarrow \{ \text{LOV'} \rightarrow \{ \text{LOT'}, \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \} \\
& \quad | ( \text{LOV' = ROV} ) \land \\
& \quad ( \text{LOT' = ROT} ) \land \\
& \quad ( \{ \text{LOK}'_k \} = \{ \text{ROK}_k \} ) \land \\
& \quad ( \{ \text{LOAC}'_c \} = \{ \text{ROAC}_c \} ) \land \\
& \quad ( \text{LOP' Meta[ RelvarCat ]} = \text{Real} ) \land \\
\end{align*}\]
(LOSM' = RealDefault)

\[
\begin{align*}
&\lor \\
&L_{OP \text{ Meta}[\text{ Exists}]} = \text{True} \land \\
&(LOT = ROT) \land \\
&(\forall kn : \text{dom}(\{LOK_k\}), \forall k : \text{ran}(\{LOK_k\}) \cdot \\
&(\text{kn} \rightarrow k) \in \{ROK_k\} \lor \\
&(\text{kn} \notin \text{dom}(\{ROK_k\}) \land k \notin \text{ran}(\{ROK_k\})) \\
&\land \\
&(\forall an : \text{dom}(\{LOAC_c\}), \forall a : \text{ran}(\{LOAC_c\}) \cdot \\
&(\text{an} \rightarrow a) \in \{ROAC_c\} \lor \\
&(\text{an} \notin \text{dom}(\{ROAC_c\}) \land a \notin \text{ran}(\{ROAC_c\}))
\end{align*}
\]

(LOP' \mapsto \{LOV' \mapsto \{LOT', \{LOK'_k\}, \{LOAC'_c\}, LOSM'\}\} | (LOV' = ROV) \land \\
(LOT' = ROT) \land \\
(\{LOK'_k\} = (\{LOK_k\} \cup \{ROK_k\}) \land \\
(\{LOAC'_c\} = (\{LOAC_c\} \cup \{ROAC_c\}) \land \\
(LOP'_{\text{Meta}[\text{RelvarCat}]} = \text{Real}) \land \\
(LOSM' = LOSM))

If LOP does not already exist, then LOP' is created with the value and type of ROP, and the keys and Ad Hoc constraints that are derivable from ROP. LOP' is specified to be of category real, and given the default physical binding of a real relvar.

If LOP does already exist, and has the same reltype as ROP, and its keys and Ad Hoc constraints are consistent with those of LOP, then LOP' takes the relvalue of ROP, retains its reltype, and includes any extra keys and Ad Hoc constraints from ROP that are not already in LOP. It continues to be of category real and with the same physical binding as before.
The ‘Stored Relvar’ Assignment

\[ LOP' \leftarrow \text{Store} \ ROP \]

Informal Description
The purpose of the \( \leftarrow \text{Store} \) assignment is to create a stored relvar with the name of \( LOP' \) and the value of \( ROP \). Any integrity constraints derived from \( ROP \) are also included in \( LOP' \). (Thus \( \leftarrow \text{Store} \) corresponds to \( \leftarrow \text{Real} \) but is for stored relvars).

If a stored relvar of the same name already exists, its value and corresponding integrity constraints are replaced by those derived from \( ROP \); any pre-existing integrity constraints that are not replaced remain, but the assigned relvalue must also satisfy them or an error results. An error also results if a relvar of the same name already exists and is not a stored relvar.

Pre Conditions

- let \( \{ \text{LeftExp} \} \rightleftharpoons \{ \text{token}_i : LOP' \text{Tokenise} \} \bullet \{ \text{LeftExp} \} \text{Meta}[ \text{Cardinality} ] = 1 \land LOP' \text{Meta}[ \text{Type} ] = \text{RelvarName} \)
  
  \( LOP' \) consists of one relvar name.

- \( ( LOP \text{Meta}[ \text{Exists} ] = \text{False} ) \lor ( ( LOP \text{Meta}[ \text{Exists} ] = \text{True} \land ( LOP \text{Meta}[ \text{RelvarCat} ] = \text{Stored} ) ) ) \)

  Either \( LOP \) does not already exist, or it already exists as a stored relvar.

- let \( \{ \text{RightExp} \} \rightleftharpoons \{ \text{token}_i : ROP \text{ Tokenise} | \text{token}_i \text{Meta}[ \text{Type} ] = \text{RelvarName} \} \bullet ( \forall \text{token}_i : \text{RightExp} \bullet ( \text{token}_i \text{Meta}[ \text{RelvarCat} ] = \text{Real} ) \lor ( \text{token}_i \text{Meta}[ \text{RelvarCat} ] = \text{Virtual} ) \lor ( \text{token}_i \text{Meta}[ \text{RelvarCat} ] = \text{Sink} ) \lor ( \text{token}_i \text{Meta}[ \text{RelvarCat} ] = \text{Stored} ) ) \)

  \( ROP \) consists of an expression such that all relvar names in it are those of real, virtual, sink or stored relvars. Note there is no requirement for any relvars to appear in the expression.

Post Conditions

- \( ( LOP \text{Meta}[ \text{Exists} ] = \text{False} \land LOP' ) \rightarrow \{ \text{LOV'} \} \rightarrow \{ \text{LOT'} : \{ \text{LOK'}_{k} \} , \{ \text{LOAC}_c \} , \text{LOSM'} \} \} \)
  
  \( | ( \text{LOV'} = \text{ROV} ) \land ( \text{LOT'} = \text{ROT} ) \land \{ \{ \text{LOK'}_{k} \} = \{ \text{ROK}_{k} \} \) \)
\[
\{( \text{LOAC}_c \} = \{ \text{ROAC}_c \} \} \land
\text{LOP}' \text{Meta}[\text{RelvarCat}] = \text{Stored} \land
\text{LOSM}' = \text{StoredDefault}
\]
\[
\lor
\]
\[
\text{LOP \text{Meta}[\text{Exists}]} = \text{True} \land
\text{LOP} = \text{ROT} \land
\forall \text{kn} : \text{dom}(\{\text{LOK}_k\}), \forall \text{k} : \text{ran}(\{\text{LOK}_k\}) \bullet
\text{(kn) } \mapsto \text{k} \in \{\text{ROK}_k\} \lor
\text{(kn} \notin \text{dom}(\{\text{ROK}_k\}) \land \text{k} \notin \text{ran}(\{\text{ROK}_k\})
\]
\[
\land
\forall \text{an} : \text{dom}(\{\text{LOAC}_c\}), \forall \text{a} : \text{ran}(\{\text{LOAC}_c\}) \bullet
\text{(an) } \mapsto \text{a} \in \{\text{ROAC}_c\} \lor
\text{(an} \notin \text{dom}(\{\text{ROAC}_c\}) \land \text{a} \notin \text{ran}(\{\text{ROAC}_c\})
\]

\[
\text{LOP}' \mapsto \{\text{LOV}' \mapsto \{\text{LOT}', \{\text{LOK}'_k\}, \{\text{LOAC}'_c\}, \text{LOSM}' \} \}
\]

\[
\mid \{\text{LOV}' = \text{ROV}\} \land
\text{LOT}' = \text{ROT} \land
\{(\{\text{LOK}'_k\} = (\{\text{LOK}_k\} \cup \{\text{ROK}_k\}) \} \land
\{(\{\text{LOAC}'_c\} = (\{\text{LOAC}_c\} \cup \{\text{ROAC}'_c\}) \} \land
\text{LOP}' \text{Meta}[\text{RelvarCat}] = \text{Stored} \land
\text{LOSM}' = \text{LOSM}
\]

If \text{LOP} does not already exist, then \text{LOP}' is created with the value and type of \text{ROP}, and the keys and Ad Hoc constraints that are derivable from \text{ROP}. \text{LOP}' is specified to be of category stored, and given the default physical binding of a stored relvar.

If \text{LOP} does already exist, and has the same reltype as \text{ROP}, and its keys and Ad Hoc constraints are consistent with those of \text{LOP}, then \text{LOP}' takes the relvalue of \text{ROP}, retains its reltype, and includes any extra keys and Ad Hoc constraints from \text{ROP} that are not already in \text{LOP}. It continues to be of category stored and with the same physical binding as before.
The ‘Relvalue’ Assignment

\[ LOP' \leftarrow ROP \]

Informal Description

The purpose of the \( \leftarrow \) assignment is to give the relvalue \( ROP \) to the relvar or unnamed view \( LOP' \).

The relvars named in \( LOP \) must already exist and may only include real, virtual or stored relvars.

The relvalue of \( LOP' \) is that of \( ROP \), and must satisfy all the integrity constraints of \( LOP \) (retained by \( LOP' \)) or an error results.

Assignments to insert, delete and amend the values of relvars are logically special cases of \( \leftarrow \).

Pre Conditions

\[ \forall \ token_t \colon LOP' \text{Tokenise} \bullet \\
( \ token_t \ Meta[ Type ] = \text{RelvarName} ) \lor \\
( \ token_t \ Meta[ Type ] = \text{RelvalueLit} ) \lor \\
( \ token_t \ Meta[ Type ] = \text{RelOp} ) \lor \\
( \ token_t \ Meta[ Type ] = \text{RelAss} ) \]

All parts of \( LOP' \) must be valid parts of a relational algebra expression, i.e. a relvar, a relvalue, a relational operator or an operator assignment.

\[ \forall \ token_t \colon LOP \text{Tokenise} \mid \ token_t \ Meta[ Type ] = \text{RelvarName} \bullet \\
\ token_t \ Meta[ Exists ] = \text{True} \]

All relvars in \( LOP \) must already exist.

\[ LOT = ROT \]

\( LOP \) and \( ROP \) have the same reltypes.

\[ \text{let } \{ A \} = \{ \forall \ token_t : LOP \text{Tokenise} | \]
\[ \ token_t \ Meta[ Type ] = \text{RelvarName} \}, \]
\[ \text{let } \{ R \} = \{ \forall \ token_t : LOP \text{Tokenise} | \]
\[ \ token_t \ Meta[ Type ] = \text{RelvarName} | \]
\[ \ token_t \ Meta[ RelvarCat ] = \text{Real} \}, \]
\[ \text{let } \{ V \} = \{ \forall \ token_t : LOP \text{Tokenise} | \]
\[ \ token_t \ Meta[ Type ] = \text{RelvarName} | \]
\[ \ token_t \ Meta[ RelvarCat ] = \text{Virtual} \}, \]
\[ \text{let } \{ S \} = \{ \forall \ token_t : LOP \text{Tokenise} | \]
\[ \ token_t \ Meta[ Type ] = \text{RelvarName} | \]
\[ \ token_t \ Meta[ RelvarCat ] = \text{Stored} \}
\[ \bullet \ ( \{ A \} = \{ R \} ) \lor ( \{ A \} = \{ V \} ) \lor ( \{ A \} = \{ S \} ) \lor \\
( \{ A \} = \{ R \} \cup \{ V \} ) \]
All relvars in \( LOP \) must together be either real, virtual, stored, or a combination of real and virtual.

**Post Conditions**

\[
LOP' \rightarrow \{ LOV' \} \rightarrow \{ LOT', \{ LOK'_k \}, \{ LOAC'_c \}, LOSM' \} \\
| \ ( LOV' = ROV ) \wedge \\
( LOT' = LOT ) \wedge \\
( \{ LOK'_k \} = \{ LOK_k \} ) \wedge \\
( \{ LOAC'_c \} = \{ LOAC_c \} ) \wedge \\
( \forall \ token'_i : LOP' \text{Tokenise} | \\
\quad \text{Meta[ Type } = \text{RelvarName }, \\
( \forall \ token_i : LOP \text{Tokenise} | \\
\quad \text{Meta[ Type } = \text{RelvarName }, \\
\quad | \ \text{token'}_i \text{Meta[ RelvarCat } = ( \text{token}_i \text{Meta[ RelvarCat ]} ) \\
\quad ) \wedge \\
( LOSM' = LOSM ) \]
\]

\( LOP' \) takes the value of \( ROP \), but retains the reltype, set of keys, set of \textit{Ad Hoc} constraints, relvar category, and binding of every relvar in \( LOP \).

**The ‘Amendment’ Assignment**

\[
LOP' \leftarrow \text{Amend[ } \{ \text{an}_j \leftarrow \text{OpExp}_j \} \}
\]

**Informal Description**

The purpose of the \( \leftarrow \text{Amend} \) assignment is to \textit{amend the values of specific attributes in every tuple of the relvar or unnamed view} \( LOP \) in order to yield \( LOP' \).
The parameter consists of a set of value assignments, where ‘an’ is the name of an attribute to receive a new value, and ‘OpExp’ is the expression to be evaluated to yield that new value. As well as value constants, ‘OpExp’ may include the names of any attributes in LOP; their values in LOP are used to derive the new attribute value.

Not all attributes of LOP need to be amended by the assignment, only those required.

<Append> changes the values of all the tuples in the left-hand operand. For this reason, the left-hand operand is often a relational operator expression that evaluates to a proper subset of a relvar, even a single tuple of a relvar, so that only that subset is amended.

The relvalue of LOP’ must satisfy all the integrity constraints of LOP (retained by LOP’) or an error results.

The relvars named in LOP must already exist and may only include real, virtual or stored relvars.

**Pre Conditions**

- \( \forall \text{token}_i : \text{LOP’ Tokenise} \quad \bullet \quad \begin{gathered} ( \text{token}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} ) \quad \lor \quad \begin{gathered} ( \text{token}_i \ \text{Meta}[\ \text{Type}] = \text{RelvalueLit} ) \quad \lor \quad \begin{gathered} ( \text{token}_i \ \text{Meta}[\ \text{Type}] = \text{RelOp} ) \quad \lor \quad ( \text{token}_i \ \text{Meta}[\ \text{Type}] = \text{RelAss} ) \end{gathered} \end{gathered} \end{gathered} \)

  All parts of LOP’ must be valid parts of a relational algebra expression, i.e. a relvar, a relvalue, a relational operator or an operator assignment.

- \( \forall \text{token}_i : \text{LOP Tokenise} \quad \text{tokens}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} \quad \bullet \quad \text{tokens}_i \ \text{Meta}[\ \text{Exists}] = \text{True} \)

  All relvars in LOP must already exist.

- \( \text{let } \{ A \} = \{ \forall \text{token}_i : \text{LOP Tokenise} | \text{tokens}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} \}, \) \( \text{let } \{ R \} = \{ \forall \text{token}_i : \text{LOP Tokenise} | \text{tokens}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} | \text{tokens}_i \ \text{Meta}[\ \text{RelvarCat}] = \text{Real} \}, \) \( \text{let } \{ V \} = \{ \forall \text{token}_i : \text{LOP Tokenise} | \text{tokens}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} | \text{tokens}_i \ \text{Meta}[\ \text{RelvarCat}] = \text{Virtual} \}, \) \( \text{let } \{ S \} = \{ \forall \text{token}_i : \text{LOP Tokenise} | \text{tokens}_i \ \text{Meta}[\ \text{Type}] = \text{RelvarName} | \text{tokens}_i \ \text{Meta}[\ \text{RelvarCat}] = \text{Stored} \} \)

  \( \bullet \quad ( \{ A \} = \{ R \} ) \quad \lor \quad ( \{ A \} = \{ V \} ) \quad \lor \quad ( \{ A \} = \{ S \} ) \quad \lor \quad ( \{ A \} = \{ R \} \ \text{Union} \ \{ V \} ) \)

  All relvars in LOP must
together be either real, virtual, stored, or a combination of real and virtual.

- $\forall \text{OpExp}_j \mid \forall \text{token}_i : (\text{OpExp}_j \text{Tokenise} ) \mid$
  - $(\text{token}_i \text{Meta}[\text{Type}] = \text{AttributeName} ) \land$
  - $(\text{token}_i \text{Meta}[\text{AttributeName}] \in (\text{LOP Meta}[\text{AttributeName}]) )$

All the attributes (denoted by their names) appearing in every expression (represented by a sequence of tokens) are in LOP.

- $\forall \text{an}_j : (\text{an}_j \text{Meta}[\text{Type}] = \text{Name} ) \land$
  - $(\text{an}_j \not\in (\text{LOP Meta}[\text{AttributeName}]) )$

All the attributes receiving a value are in LOP.

- $\{ \text{an}_j \} \subseteq (\text{LOP Meta}[\text{AttributeName}])$

The set of attributes to be amended form a subset, not necessarily a proper subset, of all those in LOP.

- $\forall j \in \{ \text{an}_j \leftarrow \text{OpExp}_j \} \mid \text{an}_j \text{Meta}[\text{AttributeType}] = (\text{OpExp}_j \text{Meta}[\text{AttributeType}])$

For all amendment assignments, the type of the attribute receiving the new value is the same as the type of the expression whose value is assigned to the attribute.

Post Conditions

- $\text{LOP}' \rightarrow \{ \text{LOV}' \rightarrow \{ \text{LOT}', \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM}' \} \}$
  - $(\forall t'_i : \text{LOV}', t_i : \text{LOV} \mid \forall a'_j : t'_i, a_j : t_i) \mid$
  - $(a'_j \text{Meta}[\text{AttributeName}] \not\in \{ \text{an}_j \} \Rightarrow (a'_j = a_j))$
  - $(a'_j \text{Meta}[\text{AttributeName}] \in \{ \text{an}_j \} \Rightarrow (a'_j = \text{OpExp}_j \text{Eval}))$
  - $(\text{LOT}' = \text{LOT}) \land$
  - $(\{ \text{LOK}'_k \} = \{ \text{LOK}'_k \}) \land$
  - $(\{ \text{LOAC}'_c \} = \{ \text{LOAC}'_c \}) \land$
  - $(\forall \text{token}_i : \text{LOP Tokenise} |$
    - $\text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName},)$
  - $(\forall \text{token}_i : \text{LOP Tokenise} |$
    - $\text{token}_i \text{Meta}[\text{RelvarCat}] = (\text{token}_i \text{Meta}[\text{RelvarCat}])$
  - $(\text{LOSM}' = \text{LOSM}))$

For every tuple in the result, LOP', every attribute that is not to be amended retains the value
it had before, in \( LOP \); every attribute that is to be amended is given the value provided by evaluating the expression assigned to its name using attribute values from \( LOP \). The reltype, set of keys, set of \textit{Ad Hoc} constraints, relvar category, and binding of every relvar in \( LOP \) remains unchanged in \( LOP' \).

The ‘Insertion’ Assignment

\( LOP' \leftarrow \text{Insert} \; ROP \)

Informal Description

The purpose of the \( \leftarrow \text{Insert} \) assignment is to \textit{insert the complete relvalue} \( ROP \) \textit{into the relvar or unnamed view} \( LOP \) in order to yield \( LOP' \).

The relvars named in \( LOP \) must already exist and may only be a real, virtual or stored relvars.

The relvalue of \( LOP' \) must satisfy all the integrity constraints of \( LOP \) (retained by \( LOP' \)) or an error results.

Pre Conditions

\( \forall \; \text{token}_i \colon LOP' \; \text{Tokenise} \cdot \)

( \( \text{token}_i \; \text{Meta}[ \text{Type} ] = \text{RelvarName} \) \( \checkmark \)
( \( \text{token}_i \; \text{Meta}[ \text{Type} ] = \text{RelvalueLit} \) \( \checkmark \)
( \( \text{token}_i \; \text{Meta}[ \text{Type} ] = \text{RelOp} \) \( \checkmark \)
( \( \text{token}_i \; \text{Meta}[ \text{Type} ] = \text{RelAss} \) \( \checkmark \))

All parts of \( LOP' \) must be valid parts of a relational algebra expression, i.e. a relvar, a relvalue, a relational operator or an operator assignment.
Semantics of Relational Algebra Assignments

20th November 2014 (9th March 2010)

David Livingstone

- $\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \bullet \text{token}_i \text{Meta}[\text{Exists}] = \text{True}$
  
  All relvars in $LOP$ must already exist.

- let $\{A\} = \{\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName}\}$,
  
  let $\{R\} = \{\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \mid \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Real}\}$,
  
  let $\{V\} = \{\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \mid \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Virtual}\}$,
  
  let $\{S\} = \{\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \mid \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Stored}\}$

- $(\{A\} = \{R\}) \lor (\{A\} = \{V\}) \lor (\{A\} = \{S\}) \lor (\{A\} = \{R\} \cup \{V\})$
  
  All relvars in $LOP$ must together be either real, virtual, stored, or a combination of real and virtual.

- $LOT = ROT$
  
  $LOP$ and $ROP$ must have the same reltypes.

- $(LOV \text{Intersect} ROV) = \emptyset$
  
  $LOP$ and $ROP$ must have disjoint relvalues.

**Post Conditions**

- $LOP' \Rightarrow \{LOP' \Rightarrow \{LOT', \{LOK'_k\}, \{LOAC'_c\}, LOSM'\}\}$

  $(LOV' = (LOV \text{Union} ROV) \land (LOT' = LOT) \land (\{LOK'_k\} = \{LOK_k\}) \land (\{LOAC'_c\} = \{LOAC_c\}) \land (\forall \text{token}_i : LOP' \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName}\),

  $(\forall \text{token}_i : LOP \text{Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName}) \land (\text{token}_i \text{Meta}[\text{RelvarCat}] = (\text{token}_i \text{Meta}[\text{RelvarCat}]) \land (LOSM' = LOSM))$

  The relvalue of $LOP'$ is the (disjoint) union of the relvalues of $LOP$ and $ROP$. $LOP'$ retains the reltype, set of keys, set of Ad Hoc constraints, relvar category, and binding of every relvar in $LOP$. 

The ‘Deletion’ Assignment

\( LOP' \gets \text{Delete} \ ROP \)

Informal Description

The purpose of the \( \text{Delete} \) assignment is to delete the complete relvalue \( ROP \) from the relvar or unnamed view \( LOP \).

The relvars named in \( LOP \) must already exist and may only involve real, virtual or stored relvars.

The value of \( LOP' \) must satisfy all the integrity constraints of \( LOP \) (retained by \( LOP' \)) or an error results.

It is often convenient to incorporate the ‘@’ pseudovariable in \( ROP \); ‘@’ stands for the entire left-hand operand, \( LOP \). For example

\[ LOP' \gets \text{Delete} \ @ \ \text{Restrict[ TrueExp ]} \]

would cause the deletion from \( LOP \) of those tuples within it that constitute the relvalue returned by the restriction operation.

\[ LOP' \gets \text{Delete} \ @ \]

would cause the entire contents of \( LOP \) to be deleted, leaving \( LOP' \) empty of tuples.

\( \text{<--Insert} \) and \( \text{<--Delete} \) are effectively mirror images of each other.

Pre Conditions

\( \forall \ token_i : \ LOP' \ \text{Tokenise} \) •

\( \begin{align*}
( token_i \ \text{Meta[ Type ]} = \text{RelvarName} ) & \lor \\
( token_i \ \text{Meta[ Type ]} = \text{RelvalueLit} ) & \lor \\
( token_i \ \text{Meta[ Type ]} = \text{RelOp} ) & \lor \\
( token_i \ \text{Meta[ Type ]} = \text{RelAss} )
\end{align*} \)

All parts of \( LOP' \) must be valid parts of a relational algebra expression, i.e. a relvar, a relvalue, a relational operator or an operator assignment.

\( \forall \ token_i : \ LOP \ \text{Tokenise} \ | \ token_i \ \text{Meta[ Type ]} = \text{RelvarName} \) •

\( token_i \ \text{Meta[ Exists ]} = \text{True} \)

All relvars in \( LOP \) must already exist.

\( \text{let} \ \{ A \} = \{ \forall \ token_i : \ LOP \ \text{Tokenise} | \ token_i \ \text{Meta[ Type ]} = \text{RelvarName} \}, \)

\( \text{let} \ \{ R \} = \{ \forall \ token_i : \ LOP \ \text{Tokenise} | \ token_i \ \text{Meta[ Type ]} = \text{RelvarName} | \ token_i \ \text{Meta[ RelvarCat ]} = \text{Real} \}, \)

\( \text{let} \ \{ V \} = \{ \forall \ token_i : \ LOP \ \text{Tokenise} | \ token_i \ \text{Meta[ Type ]} = \text{RelvarName} | \ token_i \ \text{Meta[ RelvarCat ]} = \text{Virtual} \}, \)

\( \text{let} \ \{ S \} = \{ \forall \ token_i : \ LOP \ \text{Tokenise} | \)
Token Meta[ Type ] = RelvarName |
  Token Meta[ RelvarCat ] = Stored |
- ( { A } = { R } )  ( { A } = { V } )  ( { A } = { S } ) 
  ( ( { A } = { R } Union { V } ) )

All relvars in LOP must
together be either real, virtual, stored, or a combination of real and virtual.

- LOT = ROT  
  LOP and ROP must have the
  same reltypes.

- LOV  ROV  
  The relvalue of ROP is a subset,
  not necessarily a proper subset, of LOP.

Post Conditions

- ( LOP’ Meta[ Exists ] = True ) ∧
  LOP’ )→ { LOV’ )→ { LOT, { LOKk }, { LOACc }, LOSM’ } } |
  ( LOV’ = ( LOV Diff ROV ) ∧
  ( LOT’ = LOT ) ∧
  ( { LOKk } = { LOKk } ) ∧
  ( { LOACc } = { LOACc } ) ∧
  ( ( ∀ token; : LOP Tokenise |
    token Meta[ Type ] = RelvarName ),
  ( ∀ token; : LOP Tokenise |
    token Meta[ Type ] = RelvarName ),
  | token; Meta[ RelvarCat ] = ( token Meta[ RelvarCat ] )
  ) ) ∧
  ( LOSM’ = LOSM )  
  LOP’ always continues to
  exist. Its relvalue is the relvalue of LOP minus the relvalue of ROP.
  LOP’ retains the reltype, set of keys, set of Ad Hoc constraints, relvar
category, and binding of every relvar in LOP.
The ‘Retrieval’ Assignment

\[ LOP' \leftarrow\text{Retrieve} \ ROP \]

Informal Description

The purpose of the \( \leftarrow\text{Retrieve} \) assignment is to export the relvalue of \( ROP \) to the sink named \( LOP' \) that represents a location outside the DB. If \( LOP \) is not a sink relvar, an error results.

Since retrieval is a very common assignment, to facilitate ease of use it is possible to just write the right-hand operand \( ROP \) as a RAQUEL statement; by default \( ROP \) is prefaced with \( \leftarrow\text{Retrieve} \) to a default sink relvar, and the relvalue of \( ROP \) is exported to that sink. If no default sink is available, the statement fails with an error.

Pre Conditions

- Let \( \left\{ \text{LeftExp} \right\} = \left\{ \text{token}_i : LOP' \text{ Tokenise} \right\} \bullet \left( \left\{ \text{LeftExp} \right\} \text{Meta}[\text{Cardinality}] = 1 \land LOP' \text{Meta}[\text{Type}] = \text{RelvarName} \land LOP \text{Meta}[\text{Exists}] = \text{True} \land LOP \text{Meta}[\text{RelvarCat}] = \text{Sink} \right) \)

  \( LOP' \) consists of one relvar name, which must be that of a pre-existing sink relvar.

- \( LOT \supseteq ROT \)

  The type of the sink relvar must encompass the type of the right-hand operand.

- \( \forall \text{token}_i : ROP \text{ Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \bullet \text{token}_i \text{Meta}[\text{Exists}] = \text{True} \)

  All relvars in \( ROP \) must already exist.

- Let \( \left\{ A \right\} = \left\{ \forall \text{token}_i : ROP \text{ Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \right\}, \left\{ R \right\} = \left\{ \forall \text{token}_i : ROP \text{ Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \land \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Real} \right\}, \left\{ V \right\} = \left\{ \forall \text{token}_i : ROP \text{ Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \land \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Virtual} \right\}, \left\{ S \right\} = \left\{ \forall \text{token}_i : ROP \text{ Tokenise} \mid \text{token}_i \text{Meta}[\text{Type}] = \text{RelvarName} \land \text{token}_i \text{Meta}[\text{RelvarCat}] = \text{Stored} \right\} \)

  \( \left( \left\{ A \right\} = \left\{ R \right\} \right) \lor \left( \left\{ A \right\} = \left\{ V \right\} \right) \lor \left( \left\{ A \right\} = \left\{ S \right\} \right) \lor \left( \left\{ A \right\} = \left\{ R \right\} \cup \left\{ V \right\} \right) \)
All relvars in $ROP$ must together be either real, virtual, stored, or a combination of real and virtual.

**Post Conditions**

- There are no post conditions applicable to the DB. The relvars of the DB are left unchanged. The sink relvar represents a location outside the DB. Having delivered the relvalue of the right-hand operand to the designated sink, the DBMS is not in a position to ensure that the result is appropriate or meets any required standard; this depends on the nature of the sink *per se*.

**The ‘Rename’ Assignment**

$LOP' <--\text{Rename} \ ROP$

**Informal Description**

The purpose of the $<--\text{Rename}$ assignment is to *give the relvar* $ROP$ *the new name* $LOP'$.

$ROP$ must be the name of an existing relvar (of any category), and $LOP'$ must not be the name of any existing relvar, or an error results.

**Pre Conditions**

- let $\{ RightExp \} == \{ \text{token}_i : ROP \ \text{Tokenise} \} \bullet$

  $\{ RightExp \} \ \text{Meta[Cardinality]} = 1 \ \wedge$
  
  $ROP \ \text{Meta[Type]} = \text{RelvarName} \ \wedge$
  
  $( ( ROP \ \text{Meta[RelvarCat]} = \text{Real} ) \ \vee$
  
  $( ROP \ \text{Meta[RelvarCat]} = \text{Virtual} ) \ \vee$
  
  $( ROP \ \text{Meta[RelvarCat]} = \text{Source} ) \ \vee$
  
  $( ROP \ \text{Meta[RelvarCat]} = \text{Sink} ) \ \vee$
  
  $( ROP \ \text{Meta[RelvarCat]} = \text{Stored} ) \ \wedge$
  
  $( ROP \ \text{Meta[Exists]} = \text{True} )$
**ROP** consists of one relvar name, which must be that of a pre-existing relvar of any category.

- let $\{ \text{LeftExp} \} = \{ \text{token}_i : \text{LOP}' \text{ Tokenise} \}$
  - $\{ \text{LeftExp} \} \text{ Meta[ Cardinality ]} = 1 \land$
  - $\text{LOP'} \text{ Meta[ Type ]} = \text{RelvarName} \land$
  - $\text{LOP Meta[ Exists ]} = \text{False}$
  
  $\text{LOP'}$ consists of the name of one relvar, which must not already exist.

**Post Conditions**

- $\text{LOP'} \rightarrow \{ \text{LOV'} \rightarrow \{ \text{LOT'}, \{ \text{LOK}'_k \}, \{ \text{LOAC}'_c \}, \text{LOSM'} \} \}$
  - $(\text{LOP'} = \text{ROP}) \land$
  - $(\text{LOV'} = \text{ROV}) \land$
  - $(\text{LOT'} = \text{ROT}) \land$
    - $(\{ \text{LOK}'_k \} = \{ \text{ROK}_k \}) \land$
    - $(\{ \text{LOAC}'_c \} = \{ \text{ROAC}_c \}) \land$
    - $(\text{LOP' Meta[ RelvarCat ]} = (\text{ROP Meta[ RelvarCat ]}) \land$
      - $(\text{LOSM'} = \text{ROSM})$
  
  $\text{ROP}$ continues to exist unchanged except that it takes the name $\text{LOP'}$. 

---

Page 36 of 43
The ‘Removal’ Assignment

\[ LOP' \leftarrow \text{Remove} \ ROP \]

Informal Description
The purpose of the \[ \leftarrow \text{Remove} \] assignment is similar to that of \[ \leftarrow \text{Delete} \], except that if the result of the deletion leaves \[ LOP' \] as an empty relvar, then \[ LOP' \] is removed from the DB Schema.

If \[ ROP \] evaluates to a relvalue empty of tuples, \[ LOP' \] will still be removed from the DB Schema if \[ LOP \] is empty of tuples.

\[ LOP' \leftarrow \text{Remove} \ @ \]
will always remove \[ LOP' \] from the DB Schema.

Pre Conditions
\begin{itemize}
  \item \( \forall \ token_i : LOP' \text{Tokenise} \) •
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Type} ] = \text{RelvarName} \right) \lor \left( token_i \ Meta[ \text{Type} ] = \text{RelvalueLit} \right) \lor \left( token_i \ Meta[ \text{Type} ] = \text{RelOp} \right) \lor \left( token_i \ Meta[ \text{Type} ] = \text{RelAss} \right) \)
    \end{itemize}
  All parts of \[ LOP' \] must be valid parts of a relational algebra expression, i.e. a relvar, a relvalue, a relational operator or an operator assignment.
  \item \( \forall \ token_i : LOP \text{Tokenise} \ | \ token_i \ Meta[ \text{Type} ] = \text{RelvarName} \) •
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Exists} ] = \text{True} \right) \)
    \end{itemize}
  All relvars in \[ LOP \] must already exist.
  \item \( \text{let} \ \{ A \} = \{ \forall \ token_i : LOP \text{Tokenise} | \)
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Type} ] = \text{RelvarName} \right) \) \end{itemize}
  \( \text{let} \ \{ R \} = \{ \forall \ token_i : LOP \text{Tokenise} | \)
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Type} ] = \text{RelvarName} | \right. \)
        \begin{itemize}
          \item \( \left( token_i \ Meta[ \text{RelvarCat} ] = \text{Real} \right) \)
        \end{itemize}
    \end{itemize}
  \( \text{let} \ \{ V \} = \{ \forall \ token_i : LOP \text{Tokenise} | \)
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Type} ] = \text{RelvarName} | \right. \)
        \begin{itemize}
          \item \( \left( token_i \ Meta[ \text{RelvarCat} ] = \text{Virtual} \right) \)
        \end{itemize}
    \end{itemize}
  \( \text{let} \ \{ S \} = \{ \forall \ token_i : LOP \text{Tokenise} | \)
    \begin{itemize}
      \item \( \left( token_i \ Meta[ \text{Type} ] = \text{RelvarName} | \right. \)
        \begin{itemize}
          \item \( \left( token_i \ Meta[ \text{RelvarCat} ] = \text{Stored} \right) \)
        \end{itemize}
    \end{itemize}
  \item \( \{ \{ A \} = \{ R \} \} \lor \{ \{ A \} = \{ V \} \} \lor \{ \{ A \} = \{ S \} \} \lor \{ \{ A \} = \{ R \} \ \text{Union} \ \{ V \} \} \)
  All relvars in \[ LOP \] must together be either real, virtual, stored, or a combination of real and virtual.
\end{itemize}
LOV = ROT

LOP and ROP must have the same reltypes.

LOV ⊇ ROV

The relvalue of ROP is a subset, not necessarily a proper subset, of LOP.

Post Conditions

( ( LOV' = ROV ) ⇒ ( LOP' Meta[ Exists ] = False ) ) ∨
( ( LOV' ⊇ ROV ) ⇒ ( LOP' Meta[ Exists ] = True ) ) ∧
( LOP' ) → { LOV' } → { LOT', { LOK'_k }, { LOAC'_c }, LOSM' } } |
( LOV' = ( LOV \ Diff \ ROV ) ∧ ( LOV' ⊇ Ø ) ) ∧
( LOT' = LOT ) ∧
( \{ LOK'_k \} = \{ LOK_k \} ) ∧
( \{ LOAC'_c \} = \{ LOAC_c \} ) ∧
( ( \forall \ token'_i : LOP' Tokenise |
  \token'_i Meta[ Type ] = RelvarName ),
( \forall \ token'_i : LOP Tokenise |
  \token'_i Meta[ Type ] = RelvarName ),
| \ token'_i Meta[ RelvarCat ] = ( \ token'_i Meta[ RelvarCat ] )
) ) ∧
( LOSM' = LOSM )

If the relvalue of ROP equals that of LOP, then LOP' does not exist, i.e. the left-hand operand has been removed from the DB Schema.

If the relvalue of ROP is a proper subset of that of LOP, then LOP' exists with a relvalue which is the relvalue of LOP minus that of ROP; LOP' has the reltype, set of keys, set of Ad Hoc constraints, relvar category, and binding of every relvar in LOP.
The ‘Access Control’ Assignment

\[ LOP \leftarrow \text{Access} [\{\{\text{UserSet}\}_p \leftarrow \{\text{AR AccSet}\}_p\}] \]

Informal Description

The purpose of the \( \text{Access} \) assignment is to add and/or remove access permissions to users of \( LOP \). For each assignment in the parameter, \( \{\text{UserSet}\}_p \) determines a set of users whose access permissions are being changed, and \( \{\text{AR AccSet}\}_p \) determines a set of access permissions being added or removed.

\( \text{AR} \) is an indicator that takes the value ‘+’ if \( \{\text{AccSet}\}_p \) is to be added to those of the user and ‘-’ if \( \{\text{AccSet}\}_p \) is to be removed from those of the user.

Both \( \{\text{UserSet}\}_p \) and \( \{\text{AccSet}\}_p \) may be prefixed by ‘~’ (after \( \text{AR} \) in the case of \( \{\text{AccSet}\}_p \)) to invert the set of users and access permissions respectively being dealt with by the assignment:

- \( \{\text{UserSet}\}_p \) represents an enumerated set of users who form a subset of the users registered with the DB Schema in which \( LOP \) exists.
  - ‘~ \( \{\text{UserSet}\}_p \)’ represents the users registered with the DB Schema in which \( LOP \) exists, who are not enumerated in \( \{\text{UserSet}\}_p \).

- \( \{\text{AccSet}\}_p \) is an enumerated set of access permissions to \( LOP \) that are to be added to or removed from those that a user already possesses.
  - ‘~ \( \{\text{AccSet}\}_p \)’ is that set of access permissions to \( LOP \) that the DBMS can provide that is not enumerated in \( \{\text{AccSet}\}_p \), and that are to be added to or removed from those that a user already possesses.

Access permissions are not part of the RAQUEL relational model. For this reason they do not appear in “The Semantics of Relations” document that defines relvars and relvalues in RAQUEL. They are needed as part of the security facilities that a DBMS provides to manage a DB. (Not all DBs require access permissions, e.g. some single-user DBs). The RAQUEL Access Permission Model is orthogonal to both the Logical Relational Model and the Physical Storage Model that the DBMS implements.

The RAQUEL Access Permission Model is as follows. Every combination of a relvar in a Schema and a user of that Schema is mapped to a set of access permissions that together determine how that user can access that relvar.

Each specific access permission defines a way in which a relvar can be accessed/used. The absence of a specific access permission means that a relvar cannot be accessed/used in that permitted way.

It is logically possible for a relvar-and-user combination to be mapped to an empty set of access permissions, which means that the user cannot access/use the relvar.

Formally the Access Permission Model for a DB Schema is represented by the mathematical relation ‘AccPermModel’, which is specified as follows:
AccPermModel \equiv \{ \\
\forall (r \mapsto u) : \\
(\text{DBSchemaMeta[RelvarName]} \ \text{CProd}(\text{DBSchemaMeta[Users]})) \\\n\forall \{ap\}_{r,u} : \mathcal{P}(\text{DBMSAccPerms}) \\
\bullet (r \mapsto u) \mapsto \{ap\}_{r,u} \\
\} \\
where '\mathcal{P}(\text{DBMSAccPerms})' represents the powerset of all the relvar access permissions that the DBMS can provide.

The RAQUEL DBMS uses the Access Permission Model as follows. When a relvar is created in a Schema, the creation process maps it to all the users in the Schema and maps all those combinations to an empty set of access permissions; except that the combination of the relvar and its user-creator is mapped to the full set of access permissions – this is necessary to avoid the Catch-22 of relvars being created which cannot then be used.

The addition of a user to a Schema automatically includes the mapping of all Schema relvars to that new user, and the mapping from each new relvar-and-user combination to an empty set of access permissions.

When a relvar or user is removed from a Schema, all the corresponding combinations, mappings and sets of access permissions are removed.

The orthogonality of the Access Permission Model means that no logical deductions regarding access permissions can be made when a relvar receives an assignment or when a relvalue is returned by an operator (in contrast to the logical deductions that can be made regarding relational properties).

All access permissions are determined solely by the explicit assignment of access permissions (except when relvars/users are created/removed as described above).

To fit the nature of the Access Permission Model, it is useful to translate the ‘p’ sets of assignments in the <=Access parameter to the following 2 mathematical relations:

1. AddAP \equiv \{ \forall u : (\{\{\text{UserSet}\}_p \bullet \text{AR}_p = '+'\} \ \text{Dist}[\text{Union}]) \bullet \\
(LOP'\mapsto u) \mapsto \{+\text{AP}\}_{LOP',u} \}
2. RemAP \equiv \{ \forall u : (\{\{\text{UserSet}\}_p \bullet \text{AR}_p = '-'\} \ \text{Dist}[\text{Union}]) \bullet \\
(LOP'\mapsto u) \mapsto \{-\text{AP}\}_{LOP',u} \}

‘AddAP’ is the parameter’s total set of mappings between LOP'-to-user mappings and sets of additional access permissions \{+\text{AP}\}_{LOP',u} to be assigned. ‘RemAP’ is the corresponding set of mappings for the access permissions \{-\text{AP}\}_{LOP',u} that are to be removed.

These 2 mathematical relations can then be straightforwardly used to amend the Access Permission Model that the DBMS uses to control access to relvar LOP.

In order to more easily cope with inverted sets, the actual sets of users and access permissions to be handled by the <=Access assignment, called
\{ ActualUsers \}_p and \{ ActualPerms \}_p respectively, are first derived from the 
\{ UserSet \}_p and \{ AccSet \}_p sets and their inversions.

It is convenient in the pre conditions to specify \{ UserSet \}_p and 
\{ AccSet \}_p, to specify the derivation of \{ ActualUsers \}_p and \{ ActualPerms \}_p 
from them, and finally to specify the derivation of ‘AddAP’ and ‘RemAP’ from the 
latter.

It is convenient to use ‘AddAP’, ‘RemAP’ and ‘AccPermModel’ to specify 
the post conditions.

Pre Conditions

\begin{itemize}
  \item let \{ LeftExp \} == \{ \text{token}_1 : LOP’ Tokenise \} •
    
    \{ LeftExp \} Meta[ Cardinality ] = 1 \land
    
    LOP’ Meta[ Type ] = RelvarName \land
    
    ( ( LOP Meta[ RelvarCat ] = Real ) \lor
    
    ( LOP Meta[ RelvarCat ] = Virtual ) \lor
    
    ( LOP Meta[ RelvarCat ] = Source ) \lor
    
    ( LOP Meta[ RelvarCat ] = Sink ) \lor
    
    ( LOP Meta[ RelvarCat ] = Stored ) ) \land
    
    ( LOP Meta[ Exists ] = \text{True} )

    LOP consists of one relvar 

\item \forall p • ( \{ UserSet \}_p \subseteq ( DBSchema Meta[ Users ] ) )

\quad \land

\quad ( ( \{ ActualUsers \}_p = \{ UserSet \}_p ) \lor

\quad ( \{ ActualUsers \}_p =

\quad \quad DBSchema Meta[ Users ] Diff \{ UserSet \}_p ) )

In each parameter 

assignment, all users whose access permissions are to be affected 
(i.e. the set ‘ActualUsers’) must be valid users of the DB Schema in 
which LOP exists, whether they are specified by explicitly enumerating 
the relevant users or by explicitly enumerating those who are not 
relevant users.

\item \forall p • ( \{ AccSet \}_p \subseteq \{ DBMSAccPerms \} )

\quad \land

\quad ( ( \{ ActualPerms \}_p = \{ AccSet \}_p ) ) \lor

\quad ( \{ ActualPerms \}_p =

\quad \quad \{ DBMSAccPerms \} Diff \{ UserSet \}_p )

In each parameter 

assignment, all access permissions to be added or removed must be 
valid access permissions that the DBMS can provide, whether they are 
specified by explicitly enumerating the relevant access permissions or 
by explicitly enumerating those that are not relevant access 
permissions.

\end{itemize}
AddAP \equiv \{ \forall p \bullet \text{AR} = '+' \bullet \\
\{ \forall u : \{ \text{ActualUsers} \}_p \bullet \\
\{ LOP' \mapsto u \} \mapsto \{ \text{ActualPerms} \}_p \} \text{Dist[ Union ] } \}

‘AddAP’ is the distributed union of all the parameter assignments of additional sets of access permissions, where $LOP'$ is mapped to each user before being mapped to the set of access permissions.

RemAP \equiv \{ \forall p \bullet \text{AR} = '-' \bullet \\
\{ \forall u : \{ \text{ActualUsers} \}_p \bullet \\
\{ LOP' \mapsto u \} \mapsto \{ \text{ActualPerms} \}_p \} \text{Dist[ Union ] } \}

‘RemAP’ is the distributed union of all the parameter assignments of sets of access permissions to be removed, where $LOP'$ is mapped to each user before being mapped to the set of access permissions.

\forall u : \text{ran( dom( AddAP ) ) \text{ Intersect ran( dom( RemAP ) ) } \bullet \\
\{ +AP \}_{LOP',u} \text{ Intersect } \{ -AP \}_{LOP',u} = \emptyset }

For each user who has both a set of access permissions to be added and a set to be removed, there should be no access permissions in common between these 2 sets.

\forall u : \text{ran( dom( AddAP ) ) } \bullet \\
\{ +AP \}_{LOP',u} \text{ Intersect ( AccPermModel Image[ LOP \mapsto u ] ) = } \emptyset }

For each user who has a set of access permissions to be added, that set should be disjoint with all those access permissions already in force.

\forall u : \text{ran( dom( RemAP ) ) } \bullet \\
\{ -AP \}_{LOP',u} \subseteq ( \text{AccPermModel Image[ LOP \mapsto u ] ) ) }

For each user who has a set of access permissions to be removed, that set should be already in force.

**Post Conditions**

\forall u : \text{ran( dom( AddAP ) ) } \bullet \\
\{ LOP' \mapsto u \} \mapsto \\
( \text{AccPermModel Image[ LOP \mapsto u ] } \text{ Union } \{ +AP \}_{LOP,u} )

For each user who has a set of access permissions to be added, that set is added to all those access permissions already in force.

\forall u : \text{ran( dom( RemAP ) ) } \bullet \\
\{ LOP' \mapsto u \} \mapsto \\
( \text{AccPermModel Image[ LOP \mapsto u ] } \text{ Diff } \{ -AP \}_{LOP,u} )
For each user who has a set of access permissions to be removed, that set is removed from all those access permissions already in force.

Note: In effect, the definition of the $\langle=\text{Access}\rangle$ assignment includes the definition of the Access Permission Model. This is feasible and desirable because the $\langle=\text{Access}\rangle$ assignment is the only part of the complete RAQUEL model that uses the Access Permission Model. The Meta operator may be used to return access permissions, but it reflects the semantics of the Access Permission Model through the Meta operator’s standard functionality.