BASICS: Updates
Relational Information Systems
Chapter 4.1-2
(Revised 99/10)

November 2, 1999

Copyright © 1999 Timothy Howard Merrett
Permission to make digital or hard copies of part or all of this work for personal or classroom use
is granted without fee provided that copies are not made or distributed for profit or commercial
advantage and that copies bear this notice and full citation in a prominent place. Copyright for
components of this work owned by others than T. H. Merrett must be honoured. Abstracting with
credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists,
requires prior specific permission and/or fee. Request permission to republish from: T. H. Merrett,
School of Computer Science, McGill University, fax 514 398 3883.

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

In our discussion of relations so far, everything has been functional, that is, without side-
effects or any changes to existing relations, except for the assignment operators, which replace,
or at least increment, any pre-existing relation which appears on the left of the assignment.
Even the editors are functional, because they change copies of their arguments, and this does
not affect the argument unless a subsequent assignment overwrites the original relation.

It is quite possible to have an almost completely functional database system (to permit
storage and sharing, it would have to be at least “assign-once”) but copying an entire relation
of gigabytes or more data just to change a few values in it is quite impractical. So we offer
an update-in-place syntax, which allows parts of relations to be modified without copying the
rest.

This syntax will use only the relational algebra, so that it does not introduce the concept
of a tuple to be updated. All updates will be specified by relational and domain algebra
operators, which is to say, in terms of values present in the relation to be updated.

We will work with the relations introduced in chapter 4.1-1, on QT-selectors. Initially, we
focus on Class(Item, Type), and we include the related ReClass(Item, Type). They appear
in figure 1. Each of the updates discussed below starts afresh with this value for Class. The
effects on Class of the preceding updates are supposed undone.
Class(Item  Type)  ReClass(Item  Type)
                  Yarn   A       Yarn   A
                  String B       String B
                  Ball   B       Top    A
                  Sandal C

Figure 1: Relations to Illustrate Updates

1 Additions

The following are three different ways to add one relation to another. Two of them we already know are synonyms. The third is the update syntax, and this is also a synonym. However, the latter two can both be done efficiently, in place. A smart compiler might figure out that the first can also be done in place, and so use the second or third as a faster implementation, but this would require initiative on the part of the compiler.

\[
\begin{align*}
\text{Class} & \leftarrow \text{Class ujoin where } \text{Item}="\text{Top}" \text{ in ReClass}; \\
\text{Class} & \leftarrow \text{where } \text{Item}="\text{Top}" \text{ in ReClass}; \\
\text{update } \text{Class add where } \text{Item}="\text{Top}" \text{ in ReClass};
\end{align*}
\]

The result in each case is

\[
\begin{align*}
\text{Class(Item  Type)} \\
\text{Yarn   A} \\
\text{String B} \\
\text{Ball   B} \\
\text{Sandal C} \\
\text{Top    A}
\end{align*}
\]

2 Deletions

For deletions, the update syntax is new, but again only introduces a synonym. (There is no in-place deleting assignment operator.)

\[
\begin{align*}
\text{Class} & \leftarrow \text{Class djoin ReClass}; \\
\text{update } \text{Class delete ReClass};
\end{align*}
\]

In both cases, the result is

\[
\begin{align*}
\text{Class(Item  Type)} \\
\text{String A} \\
\text{Ball   B} \\
\text{Sandal C}
\end{align*}
\]

3 Changes

This chapter is concerned primarily with changes, which are new. We work through a series of examples, starting with one which gives the full syntactic repertoire.

\[
\text{update Class change Type} \leftarrow "\text{B}" \text{ using ijoin on ReClass};
\]

2
This uses ReClass to specify which tuples of Class will change their Types to "B". Specifically, it uses Class ijoin ReClass to mark participating tuples in Class, and then updates these.

\[
\text{Class } \text{ijoin } \text{ReClass(} \text{Item} \quad \text{Type} \text{)}
\]

\[
\text{Yarn} \quad \text{A}
\]

The result is that only the tuple, (Yarn, A), is changed.

\[
\text{Class(} \text{Item} \quad \text{Type} \text{)}
\]

\[
\text{Yarn} \quad \text{B} \\
\text{String} \quad \text{A} \\
\text{Ball} \quad \text{B} \\
\text{Sandal} \quad \text{C}
\]

The syntax may be simplified by letting ijoin be the default.

\[
\text{update } \text{Class } \text{change } \text{Type}<-"\text{B}" \quad \text{using } \text{ReClass};
\]

The keyword on, or using if the join operator and on are omitted, is followed by any relational expression, making this a very powerful mode of selecting the tuples to change. For example, the same result as above, for the data shown, could be obtained from

\[
\text{update } \text{Class } \text{change } \text{Type}<-"\text{B}" \quad \text{using where } \text{Item} = "\text{Yarn}" \quad \text{in } \text{Class};
\]

(except we will find out that there are better ways than using the updated relation in the using clause).

The last update made changes to Class where the tuples entirely match those of ReClass. It is more plausible to use the Items in ReClass to identify which tuples of Class to change.

\[
\text{update } \text{Class } \text{change } \text{Type}<-"\text{B}" \quad \text{using [} \text{Item} \text{] in } \text{ReClass};
\]

\[
\text{Class } \text{ijoin [} \text{Item} \text{] in } \text{ReClass (} \text{Item} \quad \text{Type} \text{)}
\]

\[
\text{Yarn} \quad \text{A} \\
\text{String} \quad \text{A}
\]

which changes to B the type of every item of Class with a matching item in ReClass

\[
\text{Class(} \text{Item} \quad \text{Type} \text{)}
\]

\[
\text{Yarn} \quad \text{B} \\
\text{String} \quad \text{B} \\
\text{Ball} \quad \text{B} \\
\text{Sandal} \quad \text{C}
\]

This could be still a more convincing update if we could use Type in Class to replace Type in ReClass.

\[
\text{let } \text{NewType be Type;}
\]

\[
\text{update } \text{Class } \text{change } \text{Type}<- \text{NewType using [} \text{Item}, \text{NewType} \text{] in } \text{ReClass};
\]

\[
\text{Class } \text{ijoin [} \text{Item}, \text{NewType} \text{] in } \text{ReClass (} \text{Item} \quad \text{Type} \quad \text{NewType} \text{)}
\]

\[
\text{Yarn} \quad \text{A} \quad \text{A} \\
\text{String} \quad \text{A} \quad \text{B}
\]

This has changed the type of every item in Class that matches an item in ReClass, to the type given in ReClass.
\( \text{Class(} \text{Item \ Type}) \)

Yarn \ A
String \ B
Ball \ B
Sandal \ C

In the last example, \texttt{Top} got left out, because there is no matching \texttt{Top} in \texttt{Class}. Surely we would like to add this missing data.

\begin{verbatim}
let NewType be Type;
update Class change Type<-NewType using ujoin on [Item, NewType] in ReClass;
\end{verbatim}

We have explicitly put in a join operator. Since the \texttt{ijoin} cut out \texttt{Top}, we now use \texttt{ujoin}.

\begin{verbatim}
Class ujoin [Item, NewType] in ReClass
(Item  Type  NewType)
Yarn   A    A
String A    B
Ball   B    DC
Sandal C    DC
Top    DC   A
\end{verbatim}

Now we must discuss assignment using the \texttt{DC} null value. Because it is intended to have no effect on operations, it is plausible to suppose that \( X \leftarrow \texttt{DC} \) should not change \( X \). With this rule, the result is to replace the \texttt{Class} types by the \texttt{ReClass} types where there is a match, to leave the unmatched \texttt{Class} types alone, and to add the unmatched \texttt{ReClass} tuple to \texttt{Class}.

\begin{verbatim}
Class(Item  Type)
Yarn   A
String B
Ball   B
Sandal C
Top    A
\end{verbatim}

We wonder about other \( \mu \)-joins. It would seem that \texttt{rjoin} would have the same effect as \texttt{ujoin} in the above example: there would be no \texttt{Ball} and \texttt{Sandal} tuples in the join, so these would be left alone.

It also appears that there would be similar pairs for \texttt{ijoin} and \texttt{ljoin}, and for \texttt{djoin} and \texttt{sjoin}. So we should look at \texttt{djoin}.

\begin{verbatim}
update Class change Type<-"B" using djoin on ReClass;
\end{verbatim}

\begin{verbatim}
Class djoin ReClass (Item  Type)
String A
Ball   B
Sandal C
\end{verbatim}

Here, only the unmatched tuples of \texttt{Class} are changed. It is an \textit{exception} update.

\begin{verbatim}
Class(Item  Type)
Yarn   A
String B
Ball   B
Sandal B
\end{verbatim}
The only μ-join we have left out is \texttt{dljoin}, the strange sibling that is the converse of \texttt{djjoin}. Normally, it is not needed, because we can just swap the operands and use \texttt{djjoin}. But the \texttt{update} operand and the \texttt{using} operand cannot be swapped. With the above data, \texttt{dljoin} will just add \((\text{Top}, \ A)\) to \texttt{Class}: all the items in \texttt{Class} that match are excluded from the join, so their types will not be changed.

There are some degenerate special cases of the syntax when a \texttt{using} operand is not needed.

\texttt{update Class change Type<-"B";}

just replaces every type in \texttt{Class} by \texttt{B}. More usefully

\texttt{update Class change Type<- if Type="C" then "B" else Type;}

changes type \texttt{C} to \texttt{B}. Or, to go back to the example where we had \texttt{Class} as a \texttt{using} operand (and said it was inefficient)

\texttt{update Class change Type<- if Item="Yarn" then "B" else Type;}

which changes the type of \texttt{Yarn} to \texttt{B}.

The \texttt{using} operand may be any relational expression whatever.

\texttt{update Class change Type<-"B" using Supply ijoin where Floor=2 in Loc;}

\begin{tabular}{|l|l|l|l|l|l|}
\hline
\texttt{Item} & \texttt{Type} & \texttt{Comp} & \texttt{Dept} & \texttt{Vol} & \texttt{Floor} \\
\hline
\texttt{Yarn} & A & Domtex & Rug & 10 & 2 \\
\hline
\texttt{Yarn} & A & PlaySew & Rug & 17 & 2 \\
\hline
\texttt{String} & A & Domtex & Rug & 5 & 2 \\
\hline
\texttt{String} & A & PlaySew & Shoe & 5 & 2 \\
\hline
\texttt{String} & A & Shoeco & Shoe & 15 & 2 \\
\hline
\end{tabular}

giving

\texttt{Class(Item Type)}

\begin{tabular}{|l|l|}
\hline
\texttt{Yarn} & B \\
\hline
\texttt{String} & B \\
\hline
\texttt{Ball} & B \\
\hline
\texttt{Sandal} & C \\
\hline
\end{tabular}

A very powerful way of pinning down which tuples to update is given by a QT-selector.

\texttt{update Class change Type<-"B" using [Item] where \{(\#\geq 2) Comp, (\#> 1) Dept\} in Supply;}

Recall from chapter 4.1-1 that this QT-selector evaluated to \texttt{String} on the relations used in that chapter.

\texttt{Class ijoin \{("String")\} (Item Type)}

\begin{tabular}{|l|l|}
\hline
\texttt{String} & A \\
\hline
\end{tabular}

So the update changes the type of \texttt{String} to \texttt{B}.

\texttt{Class(Item Type)}

\begin{tabular}{|l|l|}
\hline
\texttt{Yarn} & A \\
\hline
\texttt{String} & B \\
\hline
\texttt{Ball} & B \\
\hline
\texttt{Sandal} & C \\
\hline
\end{tabular}
4 Updating Views

While the using operand may be any relational expression, the update operand must be an identifier, a single relational name. This is because, in general, views cannot be updated. (A view, as defined in section 1 of chapter 2.1, on the relational algebra, is an unevaluated expression.) We discuss this proposition briefly now.

It is not a new idea. Clearly arbitrary expressions cannot be assigned to, for instance. \( a < -2 \) is no problem, nor is \( a^3 < -8 \) (for real numbers). But \( a^2 < -4 \) can have two possible results for \( a \), and \( a \times b < -6 \) leaves an infinite choice for the value of \( a \), unless there were some arbitrary rule which said that the value already in \( b \) must not change, or that the statement is in error if \( b \) is uninitialized.

We already know that, in general, joins cannot be updated. Figure 10 of chapter 2.1 shows how adding a tuple to the result of a join renders it nondecomposable. So such an update cannot specify any change to the operand relations, let alone an unambiguous change.

There are exceptions to this limitation in special cases. Such special cases can often be characterized by semantic rules. For example, in \( R(A, B) \) and \( S(B, C) \), if we have the functional dependence \( B \rightarrow A \) then we may delete any tuple we like from \( RS = R \text{ ijoin } S \) and this will translate to a unique deletion in \( S \). (Such a dependence guarantees that \( RS \) is decomposable. Why?)

In the rest of this section, we show that QT-selector views are always updatable. There are two components to examine, select and project. We illustrate with the relation Responsibility of figure 3 of chapter 2.1, which we reproduce in figure 2.

First we assign to a select.

**where Agent="Raman" in Responsibility <- RamanResp;**

replaces the entire subrelation selected by the relation on the right of the assignment.

<table>
<thead>
<tr>
<th>Responsibility (Agent Item)</th>
<th>RamanResp (Agent Item)</th>
<th>NewItems Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raman Micro</td>
<td>Raman Laptop</td>
<td>Micro</td>
</tr>
<tr>
<td>Raman Terminal</td>
<td>Raman Palmtop</td>
<td>Laptop</td>
</tr>
<tr>
<td>Smith V.C.R.</td>
<td></td>
<td>Palmtop</td>
</tr>
<tr>
<td>Hung Micro</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the attributes of the relations must match. This is a departure from the lack of concern we have so far shown for type matching across assignments.

Assigning to a projection is perhaps less intuitive, but we have no choice.

**[Item] in Responsibility <- NewItems;**

must replace the set of Items for each Agent by the relation on the left of the assignment.
Putting these rules together, we can assign to a T-selector.

\[ \text{[Item] where Agent="Raman" in Responsibility <- NewItems;} \]

replaces Raman’s \textbf{Items} by the new items, and has the same effect, in this example, as assigning the selector from \textit{RamanResp}.

The same rules also permit us to assign to a QT-selector.

\[ \text{[Item] where \{(#=2)Agent\} in Responsibility <- NewItems;} \]

will replace \textbf{Micro} by \textit{NewItems} in \textit{Responsibility}.

\begin{verbatim}
Responsibility
(Agent    Item)
Raman  Micro
Raman  Laptop
Raman  Palmtop
Smith  Micro
Smith  Laptop
Smith  Palmtop
Hung   Micro
Hung   Laptop
Hung   Palmtop
\end{verbatim}

(\text{It is sometimes tricky to use QT-selectors in this way. How would we get the same effect, for this example data, as})

\textbf{where Agent="Raman" in Responsibility <- RamanResp;}

\text{or}

\[ \text{[Item] where Agent="Raman" in Responsibility <- NewItems;} \]

by using the QT-selector

\[ \text{[Agent] where \{(#=2)Item\} in Responsibility?} \]

\section{5 Updating Nested Relations}

The assignments following \textbf{change} in the syntax for updates allow arbitrary expressions of the domain algebra on the right hand side. To update nested relations, we extend these to allow arbitrary relational algebra expressions.

For example, suppose we wish to remove the information about the gender of children in \textit{Employee} of section 3 in chapter 3.1 (which we repeat below).
The update is simple.

\textbf{update Employee change} \texttt{Children<- [name, date] in Children;}

As well as replacing the value of a relational attribute, we might want just to modify it. This requires us to be able to nest update operations inside each other.

\textbf{update Employee change}

\hspace{1em} \textbf{update Children change} \texttt{sex<- if sex="F" then "female" else "male";}

\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \