BASICS: Updates

Relational Information Systems — Chapter 4.1-2 (Revised 99/10)

November 2, 1999

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 preceeding updates are supposed undone.

Class(*Item*	*Type*)	*ReClass*(*Item*	*Type*)
Yarn	A	Yarn	A
String	A	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.

Class <− *Class* **ujoin where** *Item*=`"Top"` **in** *ReClass*;

Class <+ **where** *Item*=`"Top"` **in** *ReClass*;

update *Class* **add where** *Item*=`"Top"` **in** *ReClass*;

The result in each case is

Class(*Item*	*Type*)
Yarn	A
String	A
Ball	B
Sandal	C
Top	A

2 Deletions

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

Class <− *Class* **djoin** *ReClass*;

update *Class* **delete** *ReClass*;

In both cases, the result is

Class(*Item*	*Type*)
String	A
Ball	B
Sandal	C

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.

update *Class* **change** *Type*<−`"B"` **using ijoin on** *ReClass*;

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

Class **ijoin** *ReClass*(*Item*	*Type*)
`Yarn`	`A`

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

Class(*Item*	*Type*)
`Yarn`	`B`
`String`	`A`
`Ball`	`B`
`Sandal`	`C`

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

update *Class* **change** *Type*<−`"B"` **using** *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

update *Class* **change** *Type*<−`"B"` **using where** *Item* = `"Yarn"` **in** *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 *Item*s in *ReClass* to identify which tuples of *Class* to change.

update *Class* **change** *Type*<−`"B"` **using** [*Item*] **in** *ReClass*;

Class **ijoin** [*Item*] **in** *ReClass* (*Item*	*Type*)
`Yarn`	`A`
`String`	`A`

which changes to `B` the type of every item of *Class* with a matching itemin *ReClass*

Class(*Item*	*Type*)
`Yarn`	`B`
`String`	`B`
`Ball`	`B`
`Sandal`	`C`

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

let *NewType* **be** *Type*;
update *Class* **change** *Type*<− *NewType* **using** [*Item, NewType*] **in** *ReClass*;

Class **ijoin** [*Item, NewType*] **in** *ReClass*

(*Item*	*Type*	*NewType*)
`Yarn`	`A`	`A`
`String`	`A`	`B`

This has changed the type of every item in *Class* that matches an item in *ReClass*, to the type given in *ReClass*.

Class(*Item*	*Type*)
Yarn	A
String	B
Ball	B
Sandal	C

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

> **let** *NewType* **be** *Type*;
> **update** *Class* **change** *Type*<−*NewType* **using ujoin on** [*Item, NewType*] **in** *ReClass*;

We have explicitly put in a join operator. Since the **ijoin** cut out `Top`, we now use **ujoin**.

Class **ujoin** [*Item, NewType*] **in** *ReClass*

(*Item*	*Type*	*NewType*)
Yarn	A	A
String	A	B
Ball	B	$\mathcal{DC}$
Sandal	C	$\mathcal{DC}$
Top	$\mathcal{DC}$	A

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

Class(*Item*	*Type*)
Yarn	A
String	B
Ball	B
Sandal	C
Top	A

We wonder about other μ-joins. It would seem that **rjoin** would have the same effect as **ujoin** in the above example: there would be no `Ball` and `Sandal` tuples in the join, so these would be left alone.

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

> **update** *Class* **change** *Type*<−`"B"` **using djoin on** *ReClass*;

Class **djoin** *ReClass* (*Item*	*Type*)
String	A
Ball	B
Sandal	C

Here, only the unmatched tuples of *Class* are changed. It is an *exception* update.

Class(*Item*	*Type*)
Yarn	A
String	B
Ball	B
Sandal	B

The only μ-join we have left out is **dljoin**, the strange sibling that is the converse of **djoin**. Normally, it is not needed, because we can just swap the operands and use **djoin**. But the **update** operand and the **using** operand cannot be swapped. With the above data, **dljoin** will just add (`Top, A`) to *Class*: all the items in *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 **using** operand is not needed.

update *Class* **change** *Type*<−`"B"`;

just replaces every type in *Class* by `B`. More usefully

update *Class* **change** *Type*<− **if** *Type*=`"C"` **then** `"B"` **else** *Type*;

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

update *Class* **change** *Type*<− **if** *Item*=`"Yarn"` **then** `"B"` **else** *Type*;

which changes the type of `Yarn` to `B`.

The **using** operand may be any relational expression whatever.

update *Class* **change** *Type*<−`"B"` **using** *Supply* **ijoin where** *Floor*=2 **in** *Loc*;

Class **ijoin** *Supply* **ijoin where** *Floor*=2 **in** *Loc*

(*Item*	*Type*	*Comp*	*Dept*	*Vol*	*Floor*)
Yarn	A	Domtex	Rug	10	2
Yarn	A	Playsew	Rug	17	2
String	A	Domtex	Rug	5	2
String	A	Playsew	Shoe	5	2
String	A	Shoeco	Shoe	15	2

giving

Class(*Item*	*Type*)
Yarn	B
String	B
Ball	B
Sandal	C

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

update *Class* **change** *Type*<−`"B"` **using**
[*Item*] **where** {(#≥2) *Comp*, (#> 1) *Dept*} **in** *Supply*;

Recall from chapter 4.1-1 that this QT-selector evaluated to `String` on the relations used in that chapter.

Class **ijoin** {(`"String"`)} (*Item*	*Type*)
String	A

So the update changes the type of `String` to `B`.

Class(*Item*	*Type*)
Yarn	A
String	B
Ball	B
Sandal	C

Responsibility	
(*Agent*	*Item*)
Raman	Micro
Raman	Terminal
Smith	V.C.R.
Hung	Micro

RamanResp	
(*Agent*	*Item*)
Raman	Micro
Raman	Laptop
Raman	Palmtop

NewItems
(*Item*)
Micro
Laptop
Palmtop

Figure 2: Relations to Illustrate View Updates of QT-Selectors

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 $RS = R$ **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.

Responsibility	
(*Agent*	*Item*)
Raman	Micro
Raman	Laptop
Raman	Palmtop
Smith	V.C.R.
Hung	Micro

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.

Responsibility	
(*Agent*	*Item*)
Raman	Micro
Raman	Laptop
Raman	Palmtop
Smith	Micro
Smith	Laptop
Smith	Palmtop
Hung	Micro
Hung	Laptop
Hung	Palmtop

Putting these rules together, we can assign to a T-selector.

[*Item*] **where** *Agent*="Raman" **in** *Responsibility* <− *NewItems*;

replaces Raman's *Items* by the new items, and has the same effect, in this example, as assigning the selector from *RamanResp*.

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

[*Item*] **where** {(#=2)*Agent*} **in** *Responsibility* <− *NewItems*;

will replace Micro by *NewItems* in *Responsibility*.

Responsibility	
(*Agent*	*Item*)
Raman	Micro
Raman	Laptop
Raman	Palmtop
Raman	Terminal
Smith	V.C.R.
Hung	Micro
Hung	Laptop
Hung	Palmtop

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

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

or

[*Item*] **where** *Agent*="Raman" **in** *Responsibility* <− *NewItems*;

by using the QT-selector

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

5 Updating Nested Relations

The assignments following **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 *Employee* of section 3 in chapter 3.1 (which we repeat below).

Employee

ENo	*name*	*Children*			*Training*	
		name	*date*	*sex*	*CNo*	*date*
105	John	Jane	800510	F	314	791010
		Eric	821005	M	606	810505
					714	820620
123	Anne	Maria	751112	F	315	810613
					423	820711
153	Bruce				314	791010
205	Ian	Bob	701016	M	314	791010
		Steve	750115	M		

The update is simple.

update *Employee* **change** *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.

update *Employee* **change**
    update *Children* **change** *sex*<−**if** *sex*=`"F"` **then** `"female"` **else** `"male"`;