BASICS: QT-Expressions
Relational Information Systems — Chapter 4.1-1 (Revised 99/10)

November 30, 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.

A major activity in database research and development has been the formulation of query languages. The facilities offered by query languages usually include the ability for a user, who is not particularly trained in the use of computers or in the implementation of databases, to interact directly with heir data. The interest in query languages has been twofold:

1. how great a variety of queres can a database cope with; and
2. how close can we make the query facilities to what the user is accustomed to, so as to minimize the period of special training hey must endure?

These are separate issues and are to some extent antithetical to each other. We focus on the first, since this book is intended for readers acquainted with computers. Everything we have covered so far has been concerned with *programming* for databases and secondary storage, and the present discussion is not an exception. What will be noteworthy about the QT-expressions discussed here is that a very simple extension to T-selectors offers a surprisingly complete query facility. With QT-expressions, we can subsume the query language as a simple special operator of the programming language.

QT-expressions consist of QT-selectors, QT-predicates, and QT-counters, of which QT-selectors are basic and the most important. We will develop the subject through examples, using the relations in figure 1.

Supply(*Comp*	*Dept*	*Item*	*Vol*)
Domtex	Rug	Yarn	10
Playsew	Rug	Yarn	17
Playsew	Toy	Yarn	20
Oddball	Toy	Yarn	13
Domtex	Rug	String	5
Domtex	Toy	String	2
Playsew	Toy	String	10
Playsew	Shoe	String	5
Shoeco	Shoe	String	15
Playsew	Toy	Ball	2
Oddball	Toy	Ball	2

Loc(*Dept*	*Floor*)
Rug	1
Rug	2
Shoe	2
Toy	1

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

Figure 1: Relations to Illustrate QT-Selectors

1 QT-Selectors

What QT-selectors add to T-selectors is *quantifiers*, hence the "Q". These remove from T-selectors the restriction that the selection condition must evaluate to true or false on *each* tuple of the relation individually. It introduces *aggregate* selectors, which permit a *set* of tuples to determine the truth or falsity of a selection condition.

We start with the example quantified query

find items supplied to more than two departments

[*Item*] **where** {(#> 2) *Dept*} **in** *Supply*

The phrase, (#> 2), is a *quantifier*, prescribing "there must be more than two different values of ...". So (#> 2) *Dept* states "there must be more than two different *Dept*s". The expression, #> 2, is a *quantifier predicate*, and the symbol, #, is a *quantifier symbol*. It is easiest to read the attribute as immediately following the quantifier symbol, so the quantified attribute, (#> 2) *Dept*, is pronounced "the number of *Dep*artment*s* is greater than 2".

To give the essentials of how the quantifier is used in the QT-selector, we show *Supply*, projected on *Item* and *Dept*. The columns to the right of the relation give the count, #*Dept*, for each *Item*, the value of the quantifier predicate, and finally the value of the *Item* for which the predicate is **true**.

[*Item, Dept*] **in** *Supply*

(*Item*	*Dept*)	#*Dept*	> 2?	*Item*
Yarn	Rug			
Yarn	Toy	2	f	
String	Rug			
String	Toy			
String	Shoe	3	t	String
Ball	Toy	1	f	

Horizontal lines show the groupings by *Item*. For each group, we must calculate #*Dept* and then test whether it is > 2. *Item*s that meet this test are selected: in this case, only `String`.

This process could be done with a sequence of projection, equivalence reduction, and selection, but it can also be done with a single pass following a sort to group *Dept* within *Item*. All the QT-expressions we will consider have this simple and relatively inexpensive implementation: one pass following a sort. They will all be evaluated from right to left, through the **where** clause to the projection list.

The quantifier predicate is so written so that *any* predicate involving a quantifier symbol may be used. For example

find items supplied to two or three departments

[*Item*] **where** {(#=2 **or** #=3) *Dept*} **in** *Supply*
or
[*Item*] **where** {(1 <# **or** #< 4) *Dept*} **in** *Supply*
or...

find items supplied to an odd number of departments

[*Item*] **where** {(# **mod** 2 = 0) *Dept*} **in** *Supply*

Not every QT-selector can be evaluated by a sequence of T-selectors and other operations. Here is a QT-selector which requires the special, one-pass (after sorting) implementation.

find items except those supplied in volumes of less than 10

[*Item*] **where** {(# = 0) *Vol*} *Vol* < 10 **in** *Supply*

Spelling out the implementation shows what happens. (The quantifier need not be applied to *Vol* in this query; it does not really matter what attribute it is applied to, but since *Vol* is already involved, it makes sense to use it.

[*Item, Vol*] **in** *Supply*

(*Item*	*Vol*)	*Vol*< 10?	# *Vol*	=0?	*Item*
Yarn	10	f			
Yarn	13	f			
Yarn	17	f			
Yarn	20	f	0	t	Yarn
String	2	t			
String	5	t			
String	5	t			
String	10	f			
String	15	f	2	f	
Ball	2	t			
Ball	2	t	1	f	

This shows the right-to-left evaluation when an ordinary (not quantifier) predicate is present. It also reveals that if we had done the T-selector,

[*Item*] **where** *Vol* < 10 **in** *Supply*,

first, no tuples with *Item* = `Yarn` would be left to give the answer. (However, even this query may be expressed by relational algebra other than QT-selectors. Both μ-joins and σ-joins can solve it. There is nothing new about QT-selectors, except their simple implementation and straightforward translation from natural language.)

This same approach permits multiple quantifiers:

find items supplied by at least two companies to more than one department

This does not have a completely obvious meaning, but it is easy to translate into a QT-selector.

[*Item*] **where** {(# $\geq$ 2) *Comp*, (# > 1) *Dept*} **in** *Supply*

To be sure of the meaning, we must follow through the procedure we have been developing.

[*Item, Comp, Dept*] **in** *Supply*

(*Item*	*Comp*	*Dept*)	#*Dept*	>1?	#*Comp*	≥2?	*Item*
Yarn	Domtex	Rug	1	f			
Yarn	Playsew	Rug					
Yarn	Playsew	Toy	2	t			
Yarn	Oddball	Toy	1	f	1	f	
String	Domtex	Rug					
String	Domtex	Toy	2	t			
String	Playsew	Toy					
String	Playsew	Shoe	2	t	2	t	String
Ball	Playsew	Toy					
Ball	Oddball	Toy	1	f			

But what if we turn it around?

find items supplied to more than one department by at least two companies

[*Item*] **where** {(# > 1) *Dept*, (# $\geq$ 2) *Comp*} **in** *Supply*

The procedure is different. The answer is different.

[*Item, Dept, Comp*] **in** *Supply*

(*Item*	*Dept*	*Comp*)	#*Comp*	≥2?	#*Dept*	>1?	*Item*
Yarn	Rug	Domtex					
Yarn	Rug	Playsew	2	t			
Yarn	Toy	Oddball					
Yarn	Toy	Playsew	2	t	2	t	Yarn
String	Rug	Domtex	1	f			
String	Shoe	Playsew					
String	Shoe	Shoeco	2	t			
String	Toy	Domtex					
String	Toy	Playsew	2	t	2	t	String
Ball	Toy	Oddball					
Ball	Toy	Playsew	2	t	1	f	

Is there an error here? No, changing the order of quantifiers generally changes the meaning. For example,

$$\forall x \exists y \; y > x \quad \neq \quad \exists y \forall x \; y > x$$

in the case of integers: any integer has some larger integer, but no integer is larger than all others.

There is a moral from even such a simple query. The meaning of natural language is not always obvious, until we have executed a procedure to clarify it. This rings the knell of any simple-minded "natural-language query" system. It also is hard on the notion of a

"non-procedural" language: there is *always* a procedure of some sort, and sometimes working through a procedure is essential for understanding.

The quantifier symbol, #, is sufficient for any query, when taken in combination with the domain algebra and the rest of the relational algebra. (We even saw that it adds no new functionality.) But a second quantifier symbol, •, meaning "the proportion of", is handy. In fact, introducing it parallels the development of classical logic, which has two quantifiers, existential ($\exists$) and universal ($\forall$). $\exists$ is the special case, ($\# > 0$), and we shall see that $\forall$ is ($\bullet = 1$).

find items supplied by all companies to the Toy department

[*Item*] **where** {($\bullet = 1$) *Comp*} *Dept* = `"Toy"` **in** *Supply*

i.e.

[*Item*] **where** {($\#/4 = 1$) *Comp*} *Dept* = `"Toy"` **in** *Supply*

Here, we get no *Item*s, because there are four companies, but each *Item* supplied to the `Toy` department is supplied by only two of them.

[*Item, Comp*] **where** *Dept* = `"Toy"` **in** *Supply*

(*Item*	*Comp*)	•*Dept*	=1?	*Item*
`Yarn`	`Oddball`			
`Yarn`	`Playsew`	`.5`	`f`	
`String`	`Domtex`			
`String`	`Playsew`	`.5`	`f`	
`Yarn`	`Oddball`			
`Yarn`	`Playsew`	`.5`	`f`	

Note that this formulation puts a specific interpretation on "all": "all" is relative to the relational expression that follows the **in**. The count is done, in this case, in *Supply* of all the different *Comp*anies, and this count is used to divide # to get the effect of •.

Two other possible interpretations of "all" are *not* followed. One is that "all" is relative to all possible values of the attribute used in the database. This would presuppose that there were domains, each with an explicit list of all possible values (or rule prescribing them), which we do not do. In any case, the programmer can express this meaning for "all" with a little manipulation of the relational and domain algebras.

A second unused interpretation of "all" would be to do the count of all values *after* the selection condition, if any, in the QT-selector. In the above query, "all" would mean all 3 companies (`Playsew, Oddball`, and `Domtex`) rather than all 4. Once again, not following this interpretation is a somewhat arbitrary choice, except that it is the most subtle of the interpretations and the hardest to remember; but a programmer can always force this interpretation by using # with suitable other operations.

We said above how to

find items supplied by all companies to the Toy department

so let us

find items supplied by most companies to the Toy department

[*Item*] **where** {($\bullet \geq .5$) *Comp*} **in** *Supply*

We have so far considered QT-selectors involving only the relation, *Supply*. More than one relation can be involved in a QT-expression, but they must all be suitably combined into

a relational expression.

find departments that supply no items of type B

[*Dept*] **where** {(# = 0) *Item*} *Type* = "B" **in** (*Supply* **ijoin** *Class*)

Sales **ijoin** *Class*

(*Dept*	*Item*	*Type*)	*Type*="B"?	#*Item*	=0?	*Dept*
Rug	Yarn	A	f			
Rug	String	A	f	0	t	Rug
Toy	Ball	B	t			
Toy	Yarn	A	f			
Toy	String	A	f	1	f	
Shoe	String	A	f	0	t	Shoe

(Note that if Shoe were not in *Supply*, but still remained a department, it would not appear in the answer, even though *Shoe* does sell no items, and, in particular, no items of type B.)

find departments that sell at least two thirds of the items supplied to the departments on the same floor

(We can pretend that *Sales* <− [*Dept, Item*] **in** *Supply*.)

let *Count* **be equiv max of** (**par** + **of** 1 **order** *Item* **by** *Floor*) **by** *Floor*;
([*Dept*] **where** (#/*Count*≥ 2/3) *Item* **in** (*Sales* **ijoin** *Loc*))
ijoin
([*Floor, Item, Count*] **in** (*Supply* **ijoin** *Loc*))

Sales **ijoin** *Loc*

(*Dept*	*Item*	*Floor*)
Rug	Yarn	1
Rug	Yarn	2
Toy	Yarn	1
Toy	String	1
Toy	Ball	1

Supply **ijoin** *Loc*

(*Floor*	*Item*	*Dept*)	*Count*
1	Yarn	Rug	3
1	Yarn	Toy	3
1	String	Toy	3
1	Ball	Toy	3
2	Yarn	Rug	2
2	String	Shoe	2

(...) **ijoin** [*Floor, Item, Count*] **in** (...)

(*Dept*	*Count*	*Item*	*Floor*)	#*Item*	≥2*Count*/3?	*Dept*
Rug	3	Yarn	1	1	f	
Rug	2	Yarn	2	1	f	
Toy	3	Yarn	1			
Toy	3	String	1			
Toy	3	Ball	1	3	t	Toy

This example requires some explaining. We must compare counts of two different kinds of *Item*. So while one may be counted with the quantifier symbol, #, the other requires domain algebra, and, for generality, the full idiom to count numbers of different values of an attribute. This idiom is used to count the *Item*s supplied, and so the quantifier, (#/*Count*≥ 2/3), is applied to the *Item*s sold.

It would not be enough just to compare two counts of *Item*s coming from (possibly) quite different relations: we must connect the sets of *Item*s together so that we know, for example, that we are not comparing {Widget, Gizmo, Whatchimacallit} with {Yarn,

`String, Ball`}, and, since the ratio of counts is $> 2/3$, incorrectly conclude that the first set consists of at least two thirds of the second. So we must join the two sides together on *Item*.

The join must also include *Floor*, since the query says "on the same floor". Finally, the two sides, *Sales* and *Supply*, each must have the *Floor*, both for the join and for the group-by, so they must each be joined with *Loc*.

2 QT-Predicates

Since we have a simple syntax and a one-pass implementation which so greatly extends the capabilities of relational selection, we exploit these advantages in other directions. The first is the *QT-predicate*, which simply leaves off the projection list and the **where** in order to return a Boolean value.

all companies supply volumes in excess of 10

{(• = 1) *Comp*} *Vol* > 10 **in** *Supply*

[*Comp, Vol*] **in** *Supply*

(*Comp*	*Vol*)	*Vol* > 10?	•*Comp*	> 1?
Domtex	10	f		
Domtex	5	f		
Domtex	2	f		
Playsew	17	t		
Playsew	20	t		
Playsew	10	f	0.75	f
Playsew	5	f		
Playsew	2	f		
Oddball	13	t		
Oddball	2	f		
Shoeco	15	t		

The attribute in the result of this expression has no name, just as for σ-joins of relations on the same attributes, or selectors with empty projection lists. This is consistent, since all three of these give Booleans. However, it may be useful to assign the result to a relation, which should thus have an attribute. We propose a default attribute name, *.bool*, to permit this. The programmer can soon rename it, e.g.,

let *allCompVol10* **be** *.bool*;

3 QT-Counters

The second adaptation of the QT-selector is the *QT-counter*, which permits the quantifier symbols, # and •, to quantify the answer attributes in the project list. Simple examples of each are

find the number of departments to which string is supplied

[#*Dept*] **where** *Item* = `"String"` **in** *Supply*

[*Dept* **where** *Item* = `"String"` **in** *Supply*

(*Dept*)	#*Dept*
`Rug`	
`Toy`	
`Shoe`	`3`

and

Find the proportion of departments to which string is supplied

[•*Dept*] **where** *Item* = `"String"` **in** *Supply*

[*Dept* **where** *Item* = `"String"` **in** *Supply*

(*Dept*)	#*Dept*/3
`Rug`	
`Toy`	
`Shoe`	`1.0`

Again, as in QT-predicates, the resulting attribute(s) is(are) anonymous, so there must be a default name, say *.numeric* (or, in a less flexible type system, *.intg* and *.real*). Clearly there may be more than one counted attribute in the projection list, but the resulting types must all be different or there will be ambiguities in the result. This limitation can be cleaned up if there proves to be strong need for multiply counted results.