8. Relational Calculus (Part II)

Relational Calculus, as defined in the previous chapter, provides the theoretical foundations for the design of practical data sub-languages (DSL). In this chapter, we will look at an example of one—in fact, the first practical DSL based on relational calculus—the Alpha.

Further to this, we will also look at an alternative calculus—still a relational calculus (ie. relations are still the objects of the calculus) but based on Domain Variables rather than Tuple Variables. Because of this, the relational calculus covered earlier is more accurately termed Relational Calculus with Tuple Variables. The reader will recall that Tuple Variables range over tuples of relations and were central in the formulation of inference rules and in the definition of well-formed formulae. Domain Variables, on the other hand, range over domain values rather than tuples and consequently require a different construction of well-formed formulae. We will discuss this alternative in the second part of this chapter.

8.1 The Data Sub-Language Alpha

DSL Alpha is directly based on relational calculus with tuple variables. It provides, however, additional constructions that increase the query formulation power of the language. Such constructions are in fact found in most practical DSL in use today.

8.1.1 Alpha Command

DSL Alpha is a set of Alpha commands, each taking the form:

\[ \text{Get} \ <\text{workspace}\> \ (<\text{target list}\> ) : <\text{WFF}> \]

<workspace> is an identifier or label that names a temporary working relation to hold the result of the command (similar to the named working relation in the ‘giving’ clause of relational algebra—see section 5.3). The attributes of this relation are specified by <target list> which is a list of tuple variable projections as in the previous chapter. <WFF> is of course a well-formed formulae of relational calculus that must be satisfied before the values in <target list> are extracted as a result tuple.
As an example, suppose the variable P ranges over the Product relation as shown in Figure 8-1. Then the following construction is a valid Alpha command:

\[
\text{Get } W(P.Pname) : P.Price \leq 1000
\]

The reader can see that except for the keyword ‘Get’ and the naming of the result relation (‘W’ in this example), the basic form is identical to the one used in the previous chapter, which would simply be written

\[
(P.Pname) : P.Price \leq 1000
\]

The semantics of the Alpha command is also exactly the same, except that the result is a named relation, as shown in the illustration.

### 8.1.2 Range Statement

In our exposition of relational calculus, tuple variables used in queries were introduced informally. We did this in the above example too (viz. “suppose the variable P …”). This will not do, of course, if we wish the language to be interpreted by a computer. Thus, tuple variables must be introduced and associated with the relations over which they range using formal constructions. In DSL Alpha, this is achieved by the range declaration statement, which takes the basic form:

\[
\text{Range } <\text{relation name}> <\text{variable name}>
\]

where <relation name> must name an existing relation and <variable name> introduces a unique variable identifier. The variable <variable name> is taken to range over <relation name> upon encountering such a declaration. The above example can now be written more completely and formally as:

\[
\text{Range Product P;}
\text{Get } W(P.Pname) : P.Price \leq 1000
\]

DSL Alpha statements and commands, as the above construction shows, are separated by semi-colons (‘;’).
DSL Alpha also differs from relational calculus in the way it quantifies variables. First, for a practical language, mathematical symbols like ‘∀’ and ‘∃’ need to be replaced by symbols easier to key in. DSL Alpha uses the symbols ‘ALL’ and ‘SOME’ to stand for ‘∀’ and ‘∃’ respectively. Second, rather than using the quantifiers in the <WFF> expression, they are introduced in the range declarations. Thus, the full syntax of range declarations is:

```
Range <relation name> <variable name> [ SOME | ALL ]
```

Note that the use of quantifiers in the declaration is optional. If omitted, the variable is taken to be a free variable whenever it occurs in an Alpha command.

Let us look at a number of examples.

**Query 1:** “Get the names and phone numbers of customers who live in London”

Assume the Customer relation as in Figure 8-2. This query will only need a single free variable to range over customer. The Alpha construction required is:

```
Range Customer X;
Get WA( X.Cname, X.Cphone ); X.Ccity = London
```

Figure 8-2 also highlights the tuples in Customer satisfying the WFF of the command and the associated result relation WA.

<table>
<thead>
<tr>
<th>Customer</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
<td>Ccity</td>
<td>Cphone</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8-2 Query 1**

**Query 2:** “Get the names of products bought by Customer #2”

For this query, we will need to access the Transaction relation, with records of which customer bought which product, and the Product relation, which holds the names of products. Assume these relations are as given in Figure 8-3. The object of our query is the Pname attribute of Product, thus the tuple variable for Product must necessarily be a free variable:

```
Range Product A;
```
The condition of the query requires us to look in the Transaction relation for a record of purchase by Customer #2 - as long as we can find one such record, the associated product is one that we are interested in. This is a clear case of existential quantification, and the variable introduced to range over Transaction is therefore given by:

Range Transaction B SOME;

The Alpha command for the query can now be written:

Get W ( A.Pname ): A.P# = B.P# And B.C# = 2

The associated tuples satisfying the WFF above are highlighted in the figure (the result relation is not shown).

Query 3: “Get the names and phone numbers of customers in London who bought the product VDU”

This is a more complex example that will involve three relations, as shown in Figure 8-4. The target data items are in the Customer relation (names and phone numbers). So the tuple variable assigned to it must be free:

Range Customer X;

Part of the condition specified is that the customer must live in London (ie. X.Ccity = London), but the rest of the condition (“ … who bought the product VDU”) can only be ascertained from the Transaction relation (record of purchase by some customer) and Product relation (name of product). In both these cases, we are just interested in finding one tuple from each, ie. that there exists a tuple from each relation that satisfies the query condition. Thus, the variables introduced for them are given by:

Range Transaction Y SOME;
Range Product Z SOME;

The Alpha command can now be written as:
Get W( X.Cname, X.Cphone ):
X.Ccity = London And X.C# = Y.C# And Y.P# = Z.P# And Z.Pname = VDU

Figure 8-4 highlights one instantiation of each variable that satisfies the above WFF.

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>Pname</td>
<td>C#</td>
</tr>
<tr>
<td>1</td>
<td>CPU</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>London</td>
<td>22630354</td>
</tr>
<tr>
<td>2</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>London</td>
<td>2234301</td>
</tr>
</tbody>
</table>

Note that the order of quantified variable declarations is important. The order above is equivalent to “∀P ∃T”. If variable T was declared before P, it would be equivalent to “∃T ∀P” which would mean something quite different! (see section 7.3)
This query involves only one relation: the Product relation (assume the Product relation as in the above examples). Now, the “most expensive product” is that for which every product has a price less than or equal to it. Or, in relational calculus, X is such a product provided that \( \forall Y \ X.\text{Price} \geq Y.\text{Price} \). Thus two variables are required, both ranging over Product but one of them is universally quantified:

- Range Product X;
- Range Product Y ALL;
- Get W(X.Pname): X.Price \( \geq \) Y.Price

It is perhaps interesting to note in passing that the choice by DSL Alpha designers to quantify variables at the point of declaration rather than at the point of use makes Alpha commands a little harder to read—it is not clear which variables are quantified just by looking at the Alpha command. One must search for the variable declaration to see how, if at all, it is quantified.

### 8.1.3 Additional Facilities

DSL Alpha provides additional facilities that operate on the results of its commands. While these are outside the realm of relational calculus, they are useful and practical functions that enhances the utility of the language. These facilities fall loosely under two headings: *qualifiers*, and *library functions*.

The qualifiers affect the order of presentation of tuples in the result relation, based on the ordering of values of a specified attribute in either an ascending or descending order, i.e. they may be thought of as sort functions over a designated attribute. Note that in
relational theory the order of tuples in a relation is irrelevant since a relation is a set of values. So the qualifiers affects only the presentation of a relation.

Syntactically, the qualifier is appended to the WFF and takes the following form:

\[
\{ \text{UP} | \text{DOWN} \} \text{<attribute name>}
\]

As an example, consider the requirement for the names of products bought by Customer #1 in descending order of their prices. The Alpha construction for this would be:

```
Range Product X;
Range Transaction Y SOME;
Get UWA( X.Pname, X.Price ): (X.P# = Y.P# And Y.C# = 2) DOWN X.Price
```

Figure 8-6 shows the relations highlighting tuples satisfying the WFF. It also shows the result relation UWA which can be seen to be ordered in descending order of price.

![Figure 8-6 Result of qualified command](image)

The library functions, on the other hand, derives (computes) new values from the data items extracted from the database. Another way to put this is that the result relation of the basic Alpha command is further transformed by library functions to yield the final result.

Why would we want to do this? Consider for example that we have a simple set of integers, say \{1,2,3\}. There are a variety of values we may wish to derive from it, such as

- the number of items, or cardinality, of the set (library function COUNT, ie. COUNT\{1,2,3\}=3)
- the sum of the values in the set (library function TOTAL, ie. TOTAL \{1,2,3\}=6)
- the minimum, or maximum, value in the set (library function MIN and MAX, ie. MIN \{1,2,3\} = 1, or MAX \{1,2,3\} = 3)
- the average of values in the set (library function AVERAGE, ie. AVERAGE \{1,2,3\} = 2)

Extending this idea to relations, and in particular the Alpha command, library functions are applied to attributes in the target list, taking the form:
As an example, consider the need to find the number of customers who bought the product VDU. This is quite a practical requirement to help management track how well some products are doing on the market. Pure relational calculus, however, has no facility to do this. But using the library function COUNT in DSL Alpha, we can write the following:

\[
\text{Range Transaction } T; \\
\text{Range Product } P \text{ SOME;} \\
\text{Get AAA( COUNT}(T.C#) ) : T.P# = P.P# \text{ And P.Pname = VDU}
\]

Figure 8-7 highlights the tuples satisfying the WFF and shows the result relation.

As another example, suppose we wanted to know how many products were bought by the customer Codd. The data items to answer this question are in the quantity field (Qnt) of the Transaction relation, but pure relational calculus can only retrieve the set of quantity values associated with purchases by Codd. What we need is the sum of these values. The library function TOTAL of DSL Alpha allows us to do this:

\[
\text{Range Transaction } T; \text{ Range Customer } C \text{ SOME;} \\
\text{Get BBB( TOTAL( T.Qnt ) ) : T.C# = C.C# \text{ And C.Cname = Codd}}
\]

Figure 8-8 summarises the execution of this Alpha command.
As a final remark, we note that we have only sampled a few library functions. It is not our aim to cover DSL Alpha comprehensively, but only to illustrate real DSLs based on the relational calculus, and to look at added features or facilities needed to turn them into practical languages.

8.2 Relational Calculus with Domain Variables

8.2.1 Domain Variables

As noted in the introduction, there is an alternative to using tuple variables as the basis for a relational calculus, and that is to use domain variables. Recall that a domain (see section 2.2) in the relational model refers to the current set of values of a given kind under an attribute name and is defined over all relations in the database, i.e. an attribute name denotes the same domain in whatever relation it occurs. A domain variable ranges over a designated domain, i.e. it can be instantiated to, or hold, any value from that domain.

For example, consider the domain Cname found in the Customer relation. This domain has three distinct values as shown in Figure 8-9. If we now introduced a variable, ‘Cn’, and designate it to range over Cname, then Cn can be instantiated to any of these values (the illustration shows it holding the value ‘Martin’).

As with tuple variables:

- a domain variable can hold only one value at any time
- domain variables can be introduced for any domain in the database
- more than one domain variable may be used to range over the same domain

Note also that the value of a domain variable is an atomic value, i.e. it does not comprise component values as was the case with tuple variables. Thus there is no need for any syntactic mechanism like the ‘dot notation’ to denote component atomic values of tuple variables. It also means that in constructing simple comparison expressions, domain variables appear directly without any embellishments, e.g. A > 1000, B = London, C ≤ 2000, D ≠ Paris, etc. (assuming of course that the variables A, B, C and D have been designated to range over appropriate domains).
In a relational calculus with domain variables we can write predicates of the form:

\[ \langle \text{relation name} \rangle( x_1, \ldots, x_n ) \]

where

- \( \langle \text{relation name} \rangle \) is the name of a relation currently defined in the database schema, and
- each \( x_i \) is a domain variable ranging over a domain from the intension of \( \langle \text{relation name} \rangle \)

Thus, suppose we have the situation as in Figure 8-10. It is then syntactically valid to write:

\[ \text{Customer}( A, B ) \]

as ‘Customer’ is a valid relation name, and the variables ‘A’ and ‘B’ range over domains that are in the intension of the Customer relation.

![Figure 8-10 Variables ranging over domains of a relation](image)

The meaning of such a predication can be stated as follows:

a predicate \( \langle \text{relation name} \rangle( x_1, \ldots, x_n ) \) is true for some given instantiation of each variable \( x_i \) if and only if there exists a tuple in \( \langle \text{relation name} \rangle \) that contains corresponding values of the variables \( x_1, \ldots, x_n \).

Thus, for example, \( \text{Customer}(A,B) \) is true when \( A=\text{Codd} \) and \( B=\text{London} \), since the first tuple of Customer has the corresponding values. In contrast, \( \text{Customer}(A,B) \) is false when \( A=\text{Codd} \) and \( B=\text{Paris} \), as no tuple in Customer have these values. In fact, the values that make \( \text{Customer}(A,B) \) true are:

<table>
<thead>
<tr>
<th>Name</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

that is, in relational algebra terms, a projection of \( \langle \text{relation name} \rangle \) over the domains that variables \( x_1, \ldots, x_n \) range over.

A query in relational calculus with domain variables take the form:

```
\((\text{target list}) : (\text{logical expression})\)

where
- \text{target list} is a comma-separated list of domain variable names, and
- \text{logical expression} is a truth-valued expression involving predicates and comparisons over domain variables and constants (the rules for constructing well-formed \text{logical expressions} will be detailed later).

The result of such a query is a set of instantiations of variables in \text{target list} that make \text{logical expression} true.

For example, consider the database state in Figure 8-11 and the query

\[(x,y) : (\text{Product}(x,y) \land y > 1000)\]

which can be paraphrased as “get product names and their prices for those products costing more than 1000”.

![Database state for the query “(x,y): (Product(x,y) \land y > 1000)”](image)

The only pair of \((x,y)\) instantiation satisfying \text{logical expression} in this case is \((\text{VDU},1200)\), ie. the result of the query is

\[
\begin{array}{ll}
\text{x} & \text{y} \\
\text{VDU} & 1200 \\
\end{array}
\]

Domain variables, like tuple variables, may also be quantified with either the universal or existential quantifier. Expressions involving quantified domain variables are interpreted in the same way as for quantified tuple variables (see 7.3).

Consider the query: “get the names of products bought by customer #1”. The required data items are in two relations: Product and Transaction, as follows.

<table>
<thead>
<tr>
<th>Product</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>Pname</td>
</tr>
<tr>
<td>1</td>
<td>CPU</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
</tr>
</tbody>
</table>
We can paraphrase the query to introduce variables and make it easier to formulate the correct formal query:

\[ x \text{ is such a product name if there is a product number } y \text{ for } x \text{ and there is a customer number } z \text{ that purchases } y \text{ and } z \text{ is equal to 1} \]

The phrase “x is such a product name” makes it clear that it is a variable for the ‘Pname’ domain, and as this is our target data value, x must be a free variable. The phrase “there is a product number y for x” clarifies two points: (1) that y is a variable for the P# domain, and (2) that it’s role is existential. Similarly, the phrase “there is a customer number z that purchases y” states that (1) z is a variable for the domain C#, and (2) it’s role is existential. This can now be quite easily rewritten as the formal query (assuming the variables x, y and z range over Pname, P# and C# respectively):

\[
(x) : \exists y \exists z \ (\text{Product}(x,y) \land \text{Transaction}(y,z) \land z = 1)
\]

where the subexpressions
- \text{Product}(x,y) captures the condition “there is a product number y for x”
- \text{Transaction}(y,z) captures the condition “there is a customer number z that purchases y”, and
- \(z = 1\) clearly requires that the customer number is 1

The reader should be able to work out the solution to the query as an exercise.

As a final example, consider the query: “get the names of customers who bought all types of the company’s products”. The reader can perform an analysis of this query as was done above to confirm that the relevant database state is as shown in Figure 8-12 and that the correct formal query is:

\[
(x) : \forall y \exists z \ (\text{Customer}(x,z) \land \text{Transaction}(y,z))
\]

This example illustrates a universally quantified domain variable y ranging over P#. For this query, this means that the “\text{Transaction}(y,z)” part of the logical expression must evaluate to true for every possible instantiation of y given a particular instantiation of z. Thus, when \(x = \text{Codd}\) and \(z = 1\), both Transaction(1,1) and Transaction(2,1) must
evaluate to true. They do in this case and Codd will therefore be part of the result set. But, when \( x = \text{Martin} \) and \( z = 2 \), Transaction(1,2) is true but Transaction(2,2) is not! So Martin is not part of the result set. Continuing in this fashion for every possible instantiation of \( x \) will eventually yield the full result.

### 8.2.2 Well-Formed Formula

We have not formally defined above what constitutes valid \(<\text{logical expression}>s\). We do so here, but for the sake of a uniform terminology, we will use the phrase well-formed formula (WFF) instead of \(<\text{logical expression}>\) just as we did for relational calculus with tuple variables. Thus a formal query in relational calculus with domain variables take the form:

\[(\text{<target list>}) : (\text{WFF})\]

where \(<\text{target list}>\) is a comma-separated list of free variable names, and a WFF is defined by the following rules:

1. \( P(A,…) \) is a WFF if \( P \) is a relation name and \( A,… \) is a list of free variables
2. \( A \theta M \) is a WFF if \( A \) is a variable, \( M \) is a constant or a variable, and \( \theta \in \{=, \neq, <, >, \leq, \geq\} \)
3. \( F_1 & F_2 \) and \( F_1 | F_2 \) are WFFs if \( F_1 \) and \( F_2 \) are WFFs
4. \( (F) \) is a WFF if \( F \) is a WFF
5. \( \exists x (F(x)) \) and \( \forall x (F(x)) \) if \( F(x) \) is a WFF with the variable \( x \) occurring free in it

As usual, the operator precedence for the ‘&’ and ‘|’, operators follow the standard precedence rules, ie. ‘&’ binds stronger than ‘|’. Thus,

\[
\begin{align*}
\text{‘F1 & F2 | F3’} & \equiv \text{‘(F1 & F2 ) | F3’}
\end{align*}
\]

Explicit use of parenthesis, as in rule (4) above, is required to override this default precedence. Thus if the intention is for the ‘|’ operator to bind stronger in the above expression, it has to be written as

\[
F1 & (F2 | F3)
\]