Function Rewriting for Query Optimization–A Demonstration

Simone Santini
National Center for Biomedical Research
ssantini@ncmir.ucsd.edu

Amarnath Gupta
San Diego Supercomputer Center
gupta@sdsc.edu

University of California San Diego

1 Introduction

Many modern databases are *extensible*, in the sense that they allow the user (or, more often, the database programmer) to define new data types and the algebras that operate on them [1, 2]. The data types created by the programmer are often defined as *objects* (in the programming language sense of the word) and the functions of their algebras are expressed using a suitable procedural programming language.

From the point of view of the database, in particular from that of the query optimzer, any function defined for a data type created by a programmer is *opaque* that is, the optimizer can't access anything other than the function name and its signature.

Complex data types such as images, time series, volumes, or other geometric data have often a composite structure and contain, as part of their definition, other data types that provide (partial) *representations* of the principal data type. The functions defined for the principal data type are often implemented in terms of the algebras of the data types that represent it. In another paper ([3]) we argued in favor of making this dependence explicit, so that the query optimizer can take advantage of it and to rewrite the functions and the conditions that are part of a query. Before continuing this argument, however, we should like to introduce an example to help us put the discussion on solid ground.

1.1 Our recurring example

Consider a data type containing image descriptions that we will call, quite unsurprisingly, `image`. In our data type, there are two ways of describing an image: with a *block color histogram* and with a set of *regions*. The block histogram divides the images in a number of rectangular regions arranged in a grid and, for each one of them, computes a color histogram (see figure 1). Structurally,

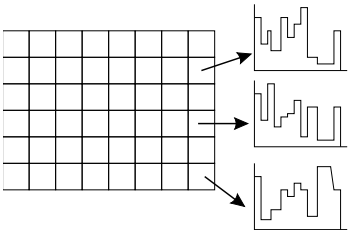

Figure 1: Block color histogram

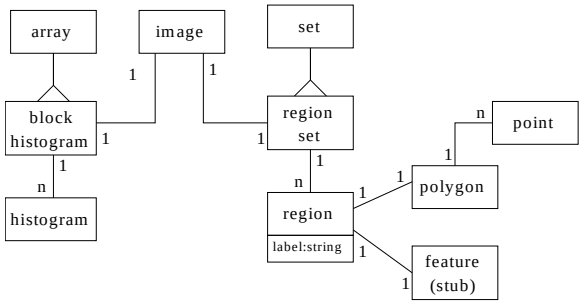

Figure 2: The `image` data type.

this description is an array of $M \times N$ objects of type `histogram`. The color histograms have, of course, their own algebras, including a function `sim` that measures the similarity of two histograms.

The second descriptor is a set of regions, obtained by applying some suitable segmentation algorithms whose details are of no concern for this paper. Each region has a boundary (represented by a polygon), a name (which is a string), and some descriptor of the image characteristics inside the region. For our examples, we don't need to specify what the region descriptors are, except to point out that they do not include color. The structure of the data type is shown in figure 2 using a standard graphical notation for class hierarchies.

Abstracting from the implementation details, each data type is defined trough its algebra and, in general, functions defined in a data type make use of functions defined for its components. Table 1 reports the functions defined on the various data types that we will use in the following examples.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

Type	Operation	Description
`Image`	`colorsim(I, l, c)`	Similarity degree between the region labeled `l` of id `I` and the color distribution (or color) `c`
	`getregions(I)`	Returns the set of labels of the regions of image `I`
	`area(I, l)`	Computes the area of the region labeled `l` in image `I`
	`gethist(I)`	returns the histogram associated to an image.
`region`	`poly(r)`	Returns the polygon defining the border of a region.
`histogram`	`count(h)`	Adds up all the entries in a histogram.
	`cut(h, p)`	Cuts the histogram, returning a histogram relative to the inside of the polygon `p`.
	`sim(h, c)`	Returns a measure of the similarity between a histogram and a color
`polygon`	`parea(p)`	measures the area inside a polygon.

Table 1: Operations defined for the image data type and for its sub-types.

A parenthetical observation should be made about the function `cut`. The histogram that describe the contents of each block do not preserve any spatial information. Consequently, it is impossible to "cut" a block histogram with a polygon (which means, in practice, to compute the histogram of the interior of the polygon) unless the polygon is contained in the grid boundaries, which is usually not the case. Therefore, an histogram that derives from the "cut" function is only an approximation of the true histogram of the interior of the polygon, and this circumstance will introduce errors in the query results.

Here, however, we will not consider this aspect of a representation, and we will always assumethat the functions are exact.

To give a flavor of how rewriting will be used, consider the following query on a database containing image data types:

```
select I
from imagetable
where count(select R from R in
         getregions(I) where P(R) ) > 3
       and
       sum(select area(R) from R in
          getregions(I) where P(R) ) > 45
```

This query asks for all images in which there are at least three regions that satisfy the predicate `P` with a total area of at least 45. One of the properties of the function `count` is that it depends only on the number of elements in a set, and not on their nature. Thanks to this observation, the first condition can be rewritten as

```
count(select area(R) from R in
   getregions(I) where P(R) ) > 3
```

Now the internal select is the same for the two conditions, and we can devise a query plan that takes advantage of this fact to re-use some of the work already done for one condition, applying it to the other.

2 A Function Algebra

If we want to allow the kind of rewriting that we are considering in this paper, it is necessary to declare explicitly to the optimizer the definitions of the functions of all the principal data types defined by the programmer (image, in this example) in terms of functions on its constituents (block histograms and region sets in the example). In other words, we need a *function algebra* to define our functions. Once the function algebra is available, we can use it to support rewriting rules. This demonstration is based on the function algebra defined in [3], whose operations are shown in figure 2 together with the programming constructs to which they give rise. Monoid homorphisms are, in essence, a form of strustural recursion and are described elsewhere [4, 5]. Primitive recursion is defined as:

$$\begin{cases} \square(f,g)(0,x) & = & f(x) \\ \square(f,g)(n+1,x) & = & g(x,\square(f,g)(n,x)) \end{cases} \quad (1)$$

These operations are used to define rewriting rules. The algebraic aspects of these rules is considered in [3]. Here, we will describe only the language based on this algebra by which the rules are specified.

3 Function algebra language

The language defined in this section specifies the rewriting rulesthat the query rewriter is allowed to use. In addition to the rules specified in this manner, there are a number of general rules that depend only on the properties of the function algebra and that apply to all functions. Examples of such rules are the distributivity of the conditional

$$\text{if}(P \circ v, u \circ f \circ v, u \circ g \circ v) \equiv u \circ \text{if}(P, f, g) \circ v \quad (2)$$

and the duplicate elimination in the cartesian composition

$$\langle\langle f, g\rangle, g\rangle \equiv \langle f \times \text{id}, \pi_2\rangle \circ \langle \text{id}, g\rangle \quad (3)$$

where π_2 is the projection function defined as $\pi_2(x, y) = y$. Some of these *unconditional* rewriting rules are reported in table 3. They are built into the optimizer and the programmer doesn't have to specify them.

Operation	Definition	Description	Programming
$f \circ g$	$(f \circ g)(x) = f(g(x))$	Function Composition	`@`
$\langle f, g \rangle$	$\langle f, g \rangle(x) = (f(x), g(x))$	cartesian composition	`comp (_, _)`
$f \times g$	$(f \times g)(x, y) = (f(x), g(y))$	cartesian product	
$[\oplus, \otimes](f)$	(see text)	monoid homomorphism	`h(op1, op2)(f)`
$\square(n, f, g)$	(see text)	primitive recursion	`nrec(n, f, g)`
$\text{if}(P, f, g)$	if P then f else g	conditional function	`if(P, f, g)`

Table 2: Operators of the function algebra.

$(f \circ g) \circ h \equiv f \circ (g \circ h)$	Associativity of composition
$\langle f \circ h, g \circ h \rangle \equiv \langle f, g \rangle \circ h$	Distributivity of composition and cartesian composition
$(f \circ h \times g \circ h) \equiv (f \times g) \circ h$	Distributivity of composition and cartesian product
$[\oplus, \otimes](f \circ g) \equiv [\oplus, \odot](f) \circ [\odot, \otimes](g)$	Associativity of homomorphism
$\text{if}(P, f \circ g, h \circ g) \equiv \text{if}(P, f, h) \circ g$	Distributivity of "if"
$\text{if}(P, g \circ f, g \circ h) \equiv g \circ \text{if}(P, f, h) \circ g$	Distributivity of "if"
$\langle \langle f, g \rangle, g \rangle \equiv \langle f \times \text{id}, \pi_2 \rangle \circ \langle \text{id}, g \rangle$	Duplicate elimination in the composition

Table 3: Unconditional rewrite rules. In the homomorphism associativity, $\odot$ is any collection monoid such that $\oplus \preceq \odot$ and $\odot \preceq \otimes$ (see [3]).

A functional specification is composed of four blocks: a *declaration* block, a *definition* block, an *inference* block, and a *rewriting* block. The declaration block contains the signatures of the functions that are implicated in the rewriting; the definition block contains the definitions of quantities and functions used in the body of the specification (a sort of "macro" block, akin to a bunch of C language "#define" statements); the inference block contains logical propositions and inference rules (in a Prolog-style notation) about function properties; the rewriting block contains the actual rewriting rules. Consider the following fragment of specification:

```
declare:
  colorsim : image, string, color -> float;
  label : region -> string;
  sim : histogram, color -> float;
  area : image, string -> float;
  hcount : histogram -> float;
  cut : histogram, poly -> histogram;
  gethist : image -> histogram;
  poly : region -> polygon;
  parea : polygon -> float;
let
  s = flatten @
       \(y, z).([set, set]
         (\x.((eq @ (id, label)))
            (x, y)(z)));
in
  inference:
# the inference block is empty
# in this case
  rewrite:
    colorsim = sim @ cut @
      comb(gethist@proj(1), poly @ s);
    area = count @ cut @
      comb(gethist@proj(1), poly @ s);
    area = parea @ polygon @ s;
end.
```

The definition of the function $\widehat{s}$ is complicate but, essentially, `s` takes the set of regions of an image and a string, and returns the region in the set whose label matches the string, if any. Note the syntax `\x.e` for λ-expressions. The rules section expands two functions defined for the image data type into their constituents. The function $\widehat{\text{proj}}(1)$ is the projection π_1, and is defined as `proj(1)(x, y) = x`.

Expanding these functions in their constituents highlighted the fact that some of them have operations in common, a fact that can be put to good use in query rewriting. The specification also tells us that the function $\widehat{\text{area}}$ can be computed in two different ways. The choice of which expansion should be used is left to the optimizer, and depends on the complete expression of the query of which the function is part.

The optimization of the example in the previous section requires a little more attention. The fact that `count(A)` doesn't change if we apply a function f to all the elements of `A` is true if `A` is a bag, but not necessarily if `A` is a set. Let $A = \{1, -1\}$, and $f(x) = x^2$, then the property would give us

$$2 = \text{count}(\{1, -1\}) = \text{count}(\{f(1), f(-1)\}) = \text{count}(\{1\}) = 1 \quad (4)$$

The property can be proven if the function f is injective or if the monoid that generates the collection is not idempotent (see [3], where this example is generalized to a whole class of functions of which `count` is an example). The functional specification for this property of `count` is the following:

```
declare:
  count : M -> int;
  label : region -> string;
  area : image, string -> float;

inference:
  is_inj(label);
  is_inj(comp(F,G)) :- is_inj(F);
  is_inj(comp(F,G)) :- is_inj(G);
  is_inj((F, G)) :- is_inj(F), is_inj(G);
  idemp(set);

rewrite:
  (not (idemp(M)) or is_inj(F)) ::-
      count = count @ [M,M](F)
end.
```

The inference block here declares that the function "label" is injective and, in addition, states some rules of injecivity, such as the fact that $\langle f, g \rangle$ is injective if either f or g is, and that $f \times g$ is injective is both f and g are. Finally, the rule states that if the monoid to which count is applied (the variable `M`) is not idempotent, or if the function `F` is injective, then `count` can be composed with an application of `F` to each element of a structure. Note the symbol `::-` which indicates a conditional rule: the rule on the left hand side can be invoked only if the condition on the right hand side is verified.

4 The Demo

The demo will show a working example of the functional rewriting system outlined in this paper, mainly using the image data type example to which we made reference here. For the purpose of the demonstration, the query rewriter will be connected to a database containing images in the required representation.

The system allows the user to register and update functional descriptions and uses them to re-write queries posed in SQL and using the functions pertinent to the data model. A specification of the cost of each function, not considered here for the sake of brevity is also part of the system, and is used for cost-based optimization.

5 Acknowledgements

The work presented in this paper was done under the auspices and with the funding of NIH project NCRR RR08 605, *Biomedical Imaging Research Network*, which the authors gratefully acknowledge.

References

[1] M. Stonebraker and L. A. Rowe, "The design of Postgres," in *ACM Conf. on SIGMOD*, pp. 340–355, June 1986.

[2] P. Seshadri, M. Livny, and R. Ramakrishnan, "E-ADTs: Turbo-charging complex data," *Data Engineering Bulletin*, vol. 19, no. 4, pp. 11–18, 1996.

[3] S. Santini and A. Gupta, "Function rewriting for query optimization," in *Proceedings of the 29th VLDB Conference (submitted)*, 2003.

[4] L. Fegaras and D. Maier, "Towards an effective calculus for object query languages," in *Proceedings of SIGMOD '95, San Jose*, pp. 47–58, 1995.

[5] P. Buneman, S. Naqvi, V. Tannen, and L. Wong, "Pronciples of programming with complex objects and collection types," *Theoretical Computer Science*, vol. 149, no. 1, pp. 3–48, 1995.