RAQUEL Token Class – Function Members

Overview

There are 2 situations in which Raquel Token objects are created :

1. The Creation of Raquel Tokens by the Compactor.
2. The Re-Writing of Raquel Algebra Expressions.

Creation by the Compactor

As with any other programming language, a RAQUEL statement first has to have a parse tree of logical tokens created in order to represent the statement in a standardised way. The parse tree then drives the execution of the statement.

RAQUEL uses a traditional Tokeniser and Parser for this, both of which operate using traditional tokenising and parsing techniques.

However because of the power and flexibility of the RAQUEL notation, RAQUEL statements also use a Compactor.

The Compactor takes the sequence of Word Tokens generated by the Tokeniser from the statement. It generates a sequence of Raquel Tokens (i.e. the logical tokens) from the Word Tokens. The Parser takes the sequence of Raquel Tokens and re-structures them into a parse tree.

The Compactor is necessary not only because some Raquel Tokens can be complex, but also because some tokens require their own internal parse tree(s) of Raquel Tokens. To achieve this, the Compactor recursively calls the Tokeniser, Compactor and Parser, to whatever finite depth of recursion is implicit in the original RAQUEL statement.

Hence it is the Compactor that creates Raquel Tokens. However the Parser also sets the values of certain data members, specifically those holding the pointer values that provide the link(s) between the token and its neighbouring Raquel Tokens in the statement's parse tree. They enable the parse tree to be traversed.

Re-Writing Raquel Algebra Expressions

Algebraic expressions can by their very nature be re-written. It is very desirable for a DBMS to be able to exploit this, and the RAQUEL DBMS is no exception.

There are a number of situations where expression re-writing is desirable. Common examples are to optimise retrieval performance, to expand default or 'short-cut' expressions into their full logical equivalent, and to expand a user-written statement into multiple statements that need to be executed in parallel so as to achieve the intent of the input statement.

The last-mentioned case is particularly relevant to the expansion of the DBMS's functional capability so that it includes the means of creating and destroying relvars in a DB.

For example, to create a new relvar, the DBMS must not only create the physical storage for it, it must also add all the necessary details about the new relvar

to the DB's Meta DB[1]. Since the Meta DB is itself a DB, additional statements must be created to insert relevant data into the relevant relvars of the Meta DB. They must be executed in parallel to the statement which physically creates the relvar.

RAQUEL also includes a **Meta** operator. It is used to get meta data about a relvar from the Meta DB. (Compared to interrogating the Meta DB directly, it provides convenience of expression, eliminates the need to know about the Meta DB, and provides security - the RAQUEL user has no ***direct*** access to the Meta DB, they use the **Meta** operator instead). So the **Meta** operator is a form of 'short-cut'. [2]

Consequently where an expression involves the **Meta** operator, that part of it needs to be re-written. The invocation of the **Meta** operator must be replaced with the relevant query expressions on the relevant relvars of the Meta DB.

There are several modules of the RAQUEL DBMS that incorporate expression re-writing as part of their implementation. Each one will operate in its own appropriate way. It will determine what Raquel Token objects need to be created, amended and/or removed, and what data member values are needed in those Raquel Token objects.

Conclusions

1. Typically a Raquel Token object will not be created all 'in one go'. It will be created in stages. Therefore it is desirable to have a Constructor function to create an 'empty' token object, and a variety of so-called ***Set*** member functions to set the values of the relevant data members.
2. Whereas the Constructor function is relatively simple, the Destructor function is relatively complex. The Destructor has to remove a token whose internal data values can vary enormously, and has to guarantee their complete removal from memory in all circumstances. (Both expression re-writing and conclusion-of-statement execution will call for the removal of tokens).
3. The Compactor and expression re-writer functions determine the values to be assigned to data members of a token. The purpose of a ***Set*** function is merely to take these value(s), supplied in its invocation, and assign them to the token's relevant data member(s).
4. There is one *partial* exception to conclusion 3, which is to do with NameLists. The values of a token's header fields and parse tree pointers are straightforward, and simple ***Set*** functions for each suffice. A token's NameList data members are designed as a single coherent storage facility that can hold a variety of types of name lists. Consequently a member function is needed to take a name list parameter, derive from it the values of the token's NameList data members, and assign those values to them. A ***SetNameLis***t member

[1] A Meta DB is sometimes also called a Data Catalog/Catalogue or a Data Dictionary.

[2] It was the necessity for Meta DB access to implement the 'Increment One' extension of the prototype RAQUEL DBMS that resulted in recognising the need to first re-factor the DBMS in order to then implement the extension.

function is needed for each type of NameList. It is convenient to also have a ***DeriveNameLists*** member function that takes a name list parameter and a 'name list type' parameter, and calls the appropriate ***SetNameLis***t function to do the work.

The Compactor calls the ***DeriveNameLists*** function. Expression re-writers may do likewise. In the latter case, it will also sometimes be useful for them to set the values of NameList data members directly, and so ***Set*** member functions are required for this purpose.

5. In order to get the values of various token data members, a comparable range of ***Get*** member functions is also needed. (This includes ***Get*** functions for NameList data members, to be used by expression re-writers).
6. The prime purpose of the member functions is to enable a token to be encapsulated. If the internal nature of the token is altered in future, they can be amended appropriately so that the changes have no affect on any software accessing a token through a member function.
7. Accessing a token includes accessing the parse trees that are internal to it. They too are accessed solely via member functions, and no other means of accessing them must be provided, to ensure token encapsulation.
8. The ***Get*** member functions must meet the range of needs created by the DBMS modules that interrogate tokens. Hence the range of ***Get*** functions will not always correlate directly to the range of ***Set*** functions.
9. Because of the variety of data that may be held in a token, it is convenient to include so-called **Query**, or ***Q***, member functions whose purpose is to check on the state of data members. This is particularly relevant to pointer data members; a ***Q*** function on a pointer would return ***true*** if it had a value and ***false*** if it had no value, i.e. had the 'nullptr' value.

1. Constructor Function

The constructor function is used to create an empty Raquel Token object; i.e. a token with a valid structure but containing no actual data.

This means that when a token is created, its data members are given default values that suit a Raquel Token, i.e. for each data member, the value it would have if it were unused in that particular token.

The default values are :

- Single character : space character value.
- Character string : empty string value.
- Pointer : C++ nullptr value (C++11 Standard).
- Vector : empty vector value.

2. Destructor Function

The destructor function will have to remove from memory :

1. The fixed set of Header Fields;
2. The set of pointers to other Raquel Tokens in the same parse tree;

3. The Parameter Expression Vector of pointers to parameter parse trees;
4. The Parameter Value Vector of pointers to literal value parameters;
5. The 3 NameList vectors, ***LeftLis***t, ***NameLists***, and ***SpecialList***.

All but the Header Fields and NameList vectors can be complex to remove, because consideration has to be given to the objects pointed at.

3. Member Functions

It is convenient to organise these around the five different groups of data members.

Token Header Fields

The values of the header fields are determined by the kind of token it is, which is declared by its name. Thus it is convenient to have one ***Set*** function that takes the name of the token as its input, and uses it to set the values of *all* the header fields.

Since typically the values of *individual* header fields are required, and the requirement varies with the circumstances, it is convenient to have a ***Get*** function for each header field.

Pointers to Neighbouring, Related Raquel Tokens

The left operand, right operand, and result pointers all have their own ***Set***, ***Get***, and ***Q*** member functions. Note that ***Set*** can be used to assign a 'nullptr' value; this is useful for correcting errors.

Parameter Expression Vector

There is a ***SetParamExp*** member function and a ***SetPosParamExp*** member function.

The former takes a valid root pointer value and adds it onto the end of the Parameter Expression Vector data member; it should be the usual function for assigning root pointer values.

The latter puts a pointer value into a position given it by a position value parameter. (Because it is a *position*, valid parameter positions start at one, not zero). It either overwrites the pointer value at that position with a new value, or, if given a 'nullptr' value, it deletes the pointer value at that position and rolls back any subsequent pointer values to ensure there are no 'gaps' in the vector.

There is a ***GetNoParamExps*** member function that returns the number of parameters that are expressions. It corresponds to the length of the Parameter Expression Vector.

There is a ***GetParamExp*** member function to obtain a root pointer value. It takes a position value parameter in order to return the relevant root pointer value.

There is no need for a ***Q*** member function because all root pointers in the Parameter Expression Vector must hold valid pointers to parse tree root tokens. ***GetNoParamExps*** will return zero if no such pointers exist.

Parameter Value Vector
There are ***SetLeftParamValue*** and ***SetRightParamValue*** member functions to assign root pointer values to the left and right parameter values respectively. (They may also be used to assign 'nullptr' values).

There are also corresponding ***Get*** member functions.

Parameter NameLists
As described above, the ***DeriveNameLists*** member function sets the values of the NameList data members, using the relevant ***SetNameLis***t member functions.

Set and ***Get*** member functions access the individual NameList data members.

The ***CreateNameListParam*** member function recreates a parameter text value from the values of the NameList data members, again using the relevant ***CreateListParam*** member function.

(N.B. Further member functions may be added in future, as required).