Extending the Compactor to Handle [IncrementOne]

Introduction

To do its job, the Compactor has to recognise a range of Word/Lex token patterns, which requires looking up the names of operators, assignments, relvars, attributes, scalar & relational types, and scalar operators & assignments.

The names of relational operators and assignments can be incorporated within the Compactor, because they are specific to the RAQUEL notation. (This is traditionally done via decision tables built into the code. If new operators or assignments are added, or others deleted or amended, this can be accomplished by changing the decision table and re-compiling the code[1]).

The names of relvars, attributes and relational types must be read in from the Meta DB, since they depend on the DB currently being accessed.

The names of scalar types, together with the names of their operators and assignments, must be read in from a third source. Apart from the primitive types of ***Truth***, ***Number***, and ***Text***, the scalar types available depend neither on the RAQUEL notation nor the DB being accessed, but on the scalar types that have been plugged in to the DBMS, whose implementation the DBMS must therefore be able to utilise[2].

Hence developing and testing the Compactor is very different from developing and testing the Tokeniser and Parser, neither of which should need alteration to handle extensions to the RAQUEL notation.

Strategy of Unit Testing

The Poc Raquel DBMS is created from a complex web of object classes. If the only way of developing and testing a developed version of the Compactor were to use it *in situ* within the DBMS, and errors arose, one could have no certainty that they arose from an error within the Compactor. They could have arisen from elsewhere in the DBMS, perhaps as a result of changing the Compactor.

Hence the strategy is to develop and test the Compactor ***in isolation from*** the rest of the Raquel DBMS.

When Compactor development and unit testing is complete, the new version of the Compactor is to be put back into the RaquelDBMS, and system tested.

Isolating the Compactor from the Rest of the RaquelDBMS

The Compactor code currently has significant links with the code of other object classes in the RaquelDBMS. See the section OO Classes Used in the document **Proof of Concept – Documentation** for information, particularly the Compactor

---

[1] In principle, other methods are also possible, such as having a separate file of data about operators and assignments that is read in by the Compactor. The file is updated as and when necessary.

[2] Logically a relational DBMS always needs the scalar ***Truth*** type in order to function correctly, with the remainder *all* being optional. In practice, ***Number***, and ***Text*** are always required and need to be incorporated into the DBMS.

document (subtitled '*CompactLexTokens*', the implementation name of the Compactor). Other relevant documents are the :

- Raquel DBMS class.
- "Token Registration" document, which references the Token Handler class and the Scalar Handler class.
- Input Stack class.
- "Compactor Versus Constructor Function" document.
- The "Token Classes Derived from 'RaquelToken'" diagram.
- "Operator and Assignment Token Classes" document.
- "The 'Base' Which Relational Operator and Assignment Token Classes All Inherit" document.

CompactLexTokens is a member function of the **CInputStack** class, and calls a number of other member functions of the **CInputStack** class to carry out parts of the compaction process. It also calls the Tokeniser, Compactor (recursively) and Parser. The latter are required so that the Compactor can handle parameters and literal values.

In fact *CompactLexTokens* utilises everything within the **CInputStack** class, with the exception of member function *CreateTree*. *CreateTree* acts like a 'main' function for the Input Stack, since all it does is successively call the Tokeniser, Compactor and Parser (checking for errors arising after each of the 3 function calls); and the Input Stack only exists for this to be carried out.

CreateTree is the only public member function of the **CInputStack** class, the remainder being protected functions.

However there is code in further parts of the RaquelDBMS which is also referenced directly or indirectly by Compactor functions, and so must be provided.

Code that is used from outside the **CInputStack** class is :

- Lex/Word Token struct *SLexToken*, and typedef *LexTokenVector* (a standard vector of *SLexToken* values), both defined in TokenTypes.h.
- Enumerated type **ELexTokenType**, from TokenTypes.h.
- Member function *AddToNameList* from **CNameListToken** class.
- Member function *AddNameList* from **CRaquelDBMS** class.
- Member function *ResolveRelationType* from class **CRaquelDBMS** (invoked by *CompactLexTokens*) returns a value of enumerated type **ERelationType** from RaquelDBMS.h.
- *RelationNameCache*, a member of class **CRaquelDBMS** (invoked by *ResolveRelationType*).
- *AttributeNameCache*, a member of class **CRaquelDBMS** (invoked by *ResolveRelationType*).
- *ScalarTypes*, a member of class **CScalarHandler** (invoked by *ResolveRelationType*).
- *IsScalarFunction*, a member of class **CScalarHandler** (invoked by *ResolveRelationType*). It calls *HasFunction*, a member of class **CScalar**, which is a pure virtual function that is implemented by each scalar class.

- *GetRelationDesc*, a member of class **CRaquelDBMS** (invoked by *ResolveRelationType*). It looks through the "Name" attribute of the Meta DB relvar "Relations".
- The "Name" attribute of the Meta DB relvar "DefaultRelation".
- *CreateRaquelTokenByType*, a member function of class **CRaquelDBMS** (invoked by *CreateChainedToken*).
- *CreateRaquelToken*, a member of class **CRaquelDBMS** – not **CRaquelToken** - and is invoked by *CreateSingleToken*. It calls *CreateToken*, a member of class **CTokenHandler**. It makes use of virtual member functions of the **CRaquelToken** class, which are overridden by the corresponding function of the relevant Raquel token class.
- *CreateRaquelTokenByType*, a member of class **CRaquelDBMS** (invoked by *CreateSingleToken*). Its sole input value is of type 'EStdToken', defined in RaquelToken.h. It calls *CreateStandardToken*, a member function of class **CTokenHandler**.
- Name Lists are implemented as objects of the **CNameListToken** class, which is descended from the **CRaquelToken** class.
- All token classes (including those that use namelists) that inherit from the **CProcessToken** class, which in turn inherits from the **CRaquelToken** class. They comprise all the relational operator and assignment token classes, plus the **CScalarToken** class (for scalar values and variables) and the **CScalarProcessToken** class (for scalar operators and assignments). Objects of these classes are Raquel tokens [3].

In summary, the software involved in the Compactor implementation, beside the **CInputStack** class, is the :

- **CRaquelDBMS** class, which uses the **Hasher** class.
- **CRaquelToken** class.
- TokenTypes.h file.
- **CTokenHandler** class.
- **CNameListToken** class.
- **CProcessToken** class.
- All relational operator and assignment token classes.
- **CScalar** class.
- **CScalarHandler** class.
- **CScalarToken** class.
- **CScalarProcessToken** class.

[3] Note that the original logical design specified a standard approach to tokens. Its physical implementation should result in a single style of token – or single token class – that handles ***all*** token values. However the prototype DBMS uses 2 token classes for all the scalar tokens, but for all the relational tokens it uses an individual token class for each kind of relational token.

Implementing the Isolation of the Compactor

In order to minimise the detailed understanding of the complex Compactor code required, it seems best to start by using the entire **CInputStack** class, with *CreateTree* as a form of 'main' program, suitably amended as required. (Note that the **CRaquelDBMS** class includes the member function *ProcessTextQuery*, whose purpose is to take a statement input to the DBMS and execute it, using the **CInputStack** class's *CreateTree* member function to produce the parse tree for execution). Amendments should include :

1. Omit calling the Parser. Call only the Tokeniser (to produce Word/Lex Tokens for the Compactor) and then the Compactor. Note that the Parser will still be called from ***within*** the Compactor to handle parameter expressions and literal relational values; but the parse trees so produced are internal to the tokens which form the parse tree that represents the statement.
2. Add user outputs to display input to the Compactor, output from the Compactor, and any other required relevant data.

There will still need to be an appropriate real 'main' program to call *CreateTree*.

There are nevertheless simplifications to the non-**CInputStack** code that can be made to isolate the Compactor code as much as possible. The remaining classes and files listed above are now considered in turn, in order to see what they might be.

Only the extension of the Compactor to handle [IncrementOne] – i.e. be able to additionally cope with the **Meta** operator and **<==Attribute**, **<==Key**, and **<--Remove** assignments – is of concern here, ***not*** the algorithms that execute the operator/assignments.

The relevant files and classes are :

- **CRaquelDBMS** class.

  A (static) object of this class forms the Raquel DBMS, so it seems sensible to retain the class. However some members of the object can be dispensed with, as they are irrelevant to testing the Compactor. They are the 'ExecutionStack', 'StorageStack', and 'ExternalInterface'. This means that the RaquelDBMS initialisation and getter & setter functions for these members can also be omitted, as can the .h and .cpp files containing the code for these object classes.

  The RaquelDBMS object includes the *RegisterTokenType* member function that 'registers' all the different kinds of token. All of them can be omitted, apart from the 4 new ones required for [IncrementOne], since the Compactor only needs these to be tested to see if it functions correctly for these 4. The object files for the omitted tokens can also be omitted.

  Likewise the RaquelDBMS object includes the *RegisterScalarType* member function that 'registers' all the different kinds of scalar types. As the Demonstration DB only uses Integer, Int32 and Text types, and there is no need to add further scalar types for [IncrementOne], the

remainder can be omitted. Their object files can also be omitted.

All the relevant '#include' statements need to be omitted from the RaquelDBMS class file.

- **CRaquelToken** class. Retain 'as is'.
- TokenTypes.h file. Retain 'as is'..
- **CTokenHandler** class. Retain 'as is'.
- **CNameListToken** class. Retain 'as is'.
- **CProcessToken** class. Retain 'as is'.
- All operator and assignment token classes. Only those pertaining to the 4 new processes of [IncrementOne].
- **CScalar** class. Retain 'as is'.
- **CScalarHandler** class. Retain 'as is'.
- **CScalarToken** class. Retain 'as is'.
- **CScalarProcessToken** class. Retain 'as is'.

Those retained 'as is' either have no code that could be omitted, or it is not worthwhile to check through them in detail to see if there is any omittable code.

Note that code relevant to the 4 new processes of [IncrementOne] is not in the current prototype, and therefore will have to be ***added***. This will include new code references and '#include' statements where relevant.

Note

As described in pages 9-12 of the Poc Implementation Documentation written by the implementers, the Compactor as implemented operates slightly differently to that specified and required :

1. The assignment prefixes ('**<--**', '**<==**', and '**==**') are treated as separate text from the assignment's keyword (e.g. **Insert**, **Remove**). At the time this was allowed, to provide additional flexibility in writing statements. This is now recognised as an error, since it is important that assignment keywords are always easily distinguished from operator keywords, by always having the formers' prefixes as an integral part of the keyword *per se*.
2. The right-hand operand of some assignments – e.g. **<--Insert** – is incorporated within the assignment's RAQUEL token. The operand should be represented by its own token.