[IncrementOne] : Implementation Notes

Introduction

It is desirable that the implementation of the **Meta** operator, and the **<==Attribute**, **<=Key**, and **<--Remove** assignments take into account the planned future development of the RAQUEL DBMS.

Both the shorter and the longer term developments require parts of the current prototype to be re-factored so as to simplify the DBMS and make its future development more feasible.

These notes explain the re-factoring relevant to the implementation of the [IncrementOne] operator and assignments, and how they affect it.

Note that the re-factoring will not be implemented as an integral part of implementing [IncrementOne]; rather it will come after its implementation. However the design and implementation of [IncrementOne], while it must fit into the existing implementation structure of the prototype, should be such as to facilitate the re-factoring afterwards as much as possible.

Background

The Re-Factoring Notes summarise the re-factoring overall and the reasons for it to be a part of the development of the DBMS.

There are 2 areas of re-factoring relevant to the [IncrementOne] development :

1. Classes Representing Tokens

Each operator and assignment has its own so called 'Token' class to represent it. The members of the class fall into one of 2 groups :

1. Members that are part of, or relate to, a genuine token that is a node in a parse tree and represents the operator/assignment in that parse tree.
2. Members that are part of, or relate to, the execution of that operator/assignment. Each operator and assignment is implemented by its own algorithm.

Looked at from a general object-oriented perspective (the DBMS is built in a pure OO fashion), this design of Token classes is not unreasonable.

However from a general DBMS perspective, and from the viewpoint of the layered architecture designed for the Raquel DBMS, the design is counter-productive and needs to be changed. Specifically the 2 groups need to be separated out as follows :

1. A class for genuine tokens, whose purpose is solely to represent operators and assignments in a parse tree.
2. The operators' and assignments' execution algorithms[1] held separately as functions to be invoked as required by other parts of the DBMS.

[1] In principle there may be more than one per operator/assignment.

The genuine token part of the current token classes will continue to be used in the Compactor and Parser of the re-factored DBMS[2]. There is no logical need for the Compactor and Parser to be altered by the change.

The components of the execution algorithms fall into different categories, which are incorporated into different parts of the DBMS. This is particularly significant for the [IncrementOne] operator and assignments, and is considered later.

2. The Meta Data Stack

The Meta DB consists of a set of relvars, and these relvars are treated and their values stored in an identical fashion to the DB's relvars. The difference between Meta DB relvars and DB relvars is that :

1. Meta DB relvars have their values retrieved indirectly via the **Meta** operator and their values updated indirectly by assignments that assign properties to DB relvars [3]. (These properties are integrity constraints and bindings, and it is the purpose of the Meta DB to hold the data describing these properties).
2. DB relvars have their values retrieved and updated directly by statements input by the user.

The planned Raquel DBMS architecture has a Meta Stack, which consists of 2 modules, the Meta Operator module and the Data Definition Module.

1. The purpose of the Meta Operator module is to translate an invocation of the **Meta** operator into relational algebra expressions to be executed on Meta DB relvars.
2. The purpose of the Data Definition module is to translate an invocation of a data property assignment into relational algebra expressions that update Meta DB relvars and update the DB itself. This can involve the execution of parallel relational statements.

In both cases, the resulting translations are then treated like any other relational expression and executed by the Execution Stack.

Thus the Meta Stack modules execute some of the algorithms necessary to implement the [IncrementOne] operator and assignments, and the Execution Stack caries out the remainder. (Non-[IncrementOne] operators/assignments are just executed by the Execution Stack).

Due to time pressures, the handling of the Meta DB was not implemented in the prototype according to the above Raquel DBMS architecture.

Instead the Meta Stack was replaced by a Meta DB Stack, which acts as a scaffold. (The prototype already has a Storage Stack for the DB relvars). Because of the prototype's limitations, the Meta DB Stack only needs to provide a means of retrieving meta data about relvars, and not updating it, and is sufficient for this purpose. The

[2] The genuine token part of the current token classes does include complications which appear unnecessary, giving an opportunity to improve the design of the genuine tokens at a later date. Since the new tokens would continue to be expressed as object values, this would not require any changes to the Compactor or Parser.

[3] **<==Attribute**, **<=Key**, and **<--Remove** assignments are all examples of these.

retrieve-only usage resulted in the Meta DB Scaffold being treated like a separate Storage Stack.

This is despite the fact that the purpose of a Storage Stack is to provide a specific physical storage mechanism, which may used to physically store the value of any relvar, regardless of whether it is a Meta DB relvar or a DB relvar. In principle the DBMS may have several Storage Stacks, each one providing it with a particular physical storage mechanism. Currently the prototype has only one physical storage mechanism, i.e. it logically only has one Storage Stack, used by the DBMS to access the values of DB relvars and via the Meta DB Stack scaffold to access the values of Meta DB relvars.

The planned DBMS architecture is described in The Logical Architecture of the RAQUEL DBMS document. For comparison, the prototype's architecture is shown on page 6 of the Raquel Prototype (2009) : Implementation Documentation . The prototype's architecture needs to evolve towards the originally planned architecture.

Executing Operators and Assignments

Current Prototype

The execution members of the 'Token' classes in the current prototype operate as follows.

Each class has an *ExecuteTree* method, which handles general parse tree processing with respect to the operator/assignment (e.g. checking for valid operands). *ExecuteTree* then calls a second method, named after the particular operator/assignment, which executes the specific algorithm that implements that operator/assignment.

This 2-part approach is not used consistently for all operators/assignments. Sometimes there is just an *ExecuteTree* method, which also incorporates the implementation algorithm for the operator/assignment.

There always needs to be an *ExecuteTree* method for each operator/assignment. This is because the parse tree is executed by first executing the operator/assignment that is the root node of the parse tree, which then recursively calls *ExecuteTree* for each sub-node in the tree which does not yet have a value. *ExecuteTree* is a standardised name to allow this recursion process to apply all the way down to the leaf nodes of the parse tree. For further details, see section "***5.2) Execution***" of the Raquel Prototype (2009) : Implementation Documentation .

Re-Factored DBMS

The re-factored DBMS will have an Execution Stack whose purpose is to execute the *final* parse tree. By *final* is meant the parse tree that results after various processing has been carried out on the original parse tree, which was a direct representation of the user's input statement. See The Logical Architecture of the RAQUEL DBMS document for details.

The Execution Stack will carry out the processing that the current *ExecuteTree* method carries out, and call the relevant function (there could be a choice) to execute the operator/assignment.

Therefore it is important for the implementation of [IncrementOne] that there is always a function separate from *ExecuteTree* that executes the operator/assignment, so that it may be easily transferred ‘as is’ (or with the minimum of amendment) to the Execution Stack when the Token Separation re-factoring is carried out.

Meta DB Storage
Inherent to the [IncrementOne] implementation is the accessing of Meta DB relvars. The methods used for this will remain unchanged from the current prototype. No new Storage Stack is planned for the [IncrementOne] implementation, so no change is called for.

Implementing to Facilitate Re-Factoring

Consequently the aspect of [IncrementOne] implementation that facilitates later re-factoring consists of :

1. Creating a new ‘Token’ class for each of the **Meta** operator and the **<==Attribute**, **<=Key**, and **<--Remove** assignments, with the same general class structure and set of data and function members as the ‘Token’ classes of the existing operators and assignments. In this way they will fit into the current OO architecture of the DBMS, and be usable in the same way.
2. Implementing the *ExecuteTree* method such that its component parts may be later re-used with minimal re-programming in the re-factored DBMS.

[IncrementOne] Operators and Assignments

These differ from the operators and assignments already implemented as follows.

Every non-[IncrementOne] operator/assignment can be directly implemented by an algorithm, expressed in code as a function, that operates directly on the value(s) of its relational operand(s). The operator/assignment and its operand(s) may be considered as a small parse tree consisting of the operator/assignment node and its operand node(s). That small parse tree is generally a component of a larger parse tree that expresses the user’s complete input statement. (This does not prevent the small parse tree from being the entire tree).

By contrast, the **Meta** operator and **<==Attribute**, **<=Key**, and **<--Remove** assignments require that the small parse tree comprising them and their operand(s) be translated into different parse trees, and those new parse trees be executed. The translations are a direct logical consequence of their semantics and so must be implemented. Consider each operator/assignment in turn :

Meta operator. To execute a **Meta** operation requires translation into a parse tree describing a **Restrict** operation on the relevant Meta DB relvar, and a **Project** operation on the results of the **Restrict**ion. The latter parse tree is then executed.
Project and **Restrict** are both already implemented and have algorithms that directly execute their functionality. Since the functionality of the **Meta** operator is defined in terms of **Project** and **Restrict**, it is necessary to execute the **Meta** operator in terms of them and not via some other algorithm. This also provides simpler code.

<==Attribute assignment. The basic functionality of the assignment is to assign a set of attributes to a relvar. This requires a translation of the **<==Attribute** assignment parse tree into an **<--Insert** assignment parse tree that will add the necessary attribute data into the *Attributes* Meta DB relvar. The latter parse tree is then executed.

Depending on the definitional state of the relvar being assigned attributes, the assignment may have additional tasks to accomplish.

If the relvar does not yet exist and so is being created by the assignment, then it will also have to create the relvar by inserting the relevant data into the *Relations* Meta DB relvar. This requires the generation and execution of a parse tree. A default Candidate Key must also be given to the relvar, requiring another parse tree to be generated and executed to put the relevant data into the *Keys* Meta DB relvar. Finally an empty relvalue must be created and stored in the default storage location using the default storage mechanism[4], ready for a relvalue to be assigned to it.

If the relvar does already exist, then it must check that all attributes currently referenced by the relvar's properties are included among those being assigned to the relvar. If this is not the case, then the relevant error must be flagged and no assignment executed. (Hence this validation check must come first).

<==Key assignment. The basic functionality of the assignment is to add and/or remove a set of Candidate Keys to a relvar. This requires a translation of the **<==Key** assignment parse tree into an **<--Insert** and/or **<--Delete** assignment parse tree(s) that will add and/or remove the necessary key data to/from the *Keys* Meta DB relvar. The latter parse tree(s) is/are then executed.

If the relvar currently has only a default key, then this must be removed when using **<==Key** to assign a key. This should be incorporated into the above assignments.

Depending on the definitional state of the relvar being assigned attributes, the assignment may have additional tasks to accomplish.

If the relvar does not yet exist and is being created by the assignment, then it will also have to create the relvar by inserting the relevant data into the *Relations* Meta DB relvar, requiring another parse tree to be generated and executed. No empty relvalue is created and stored, because this is logically impossible since all the attributes are not yet known.

If the relvar does already exist and attributes have been assigned to it, then it must check that the relvar already has all the attributes in the to-be-assigned keys. If this is not the case, then the relevant error must be flagged and no assignment executed. (Hence this validation check must come first).

<--Remove assignment. The basic purpose of the assignment is to remove a relvar from the DB, but only if that relvar's value becomes an empty tuple set after the operand relvalue is deleted from it.

Thus to execute **<--Remove** requires first translating its parse tree into a corresponding **<--Delete** operation on the relvar, and executing it.

The value of the relvar is then checked. If it is not empty, the assignment

[4] The default is currently the only storage location and storage mechanism provided by the prototype.

terminates.

If the value of the relvar is empty, then the relvar is removed from the DB. This requires removing the storage of the empty relvalue from the DB[5], together with the following :

- Generate a **<--Delete** parse tree that removes the relvar from the *Relations* Meta DB relvar.
- Generate a **<--Delete** parse tree that removes the relvar from the *Attributes* Meta DB relvar.
- Generate a **<--Delete** parse tree that removes the relvar from the *Keys* Meta DB relvar[6].
- Execute the above parse trees.

The *ExecuteTree* Method in [IncrementOne] Operators and Assignments

To ensure that they fit into the current prototype and be executed, the *ExecuteTree* method of their classes must continue to be a public member function whose input is a pointer to its operand (of class **CTable**) and whose output is a Boolean value. The output must be ***True*** if the operator/assignment executes correctly, and ***False*** otherwise.

All the *ExecuteTree* methods should call a function, named after the operator/assignment in question, to actually execute the operator/assignment. For generality, let that function be called *fn*. *fn* should be a private member function of the 'Token' class. *ExecuteTree* must continue to handle the result and consequences of *fn*'s execution.

Internally *fn* should carry out all the required validation checks and other algorithmic requirements, translate the input parse tree into the required parse tree(s), and then execute those parse tree(s). In doing so, it must not re-invent algorithms to execute other operators/assignments that are invoked by the parse trees, but use their *ExecuteTree* method(s).

In order to translate an [IncrementOne] operator/assignment into a parse tree(s), it may be necessary to develop purely internal operators/assignments, i.e. an operator/assignment that does not appear in RAQUEL, but is required for the effective execution of the *final* parse tree. For example, it might be useful in the implementation of **<--Remove** to have an operator that checks for an empty relvalue. This would correspond to the "Intermediate Nodes" used in the internal parse trees of the BS12 Relational Algebra DBMS – see slide 30 of the Business System 12 presentation.

[5] A relvalue that has no tuples is a legitimate relvalue.

[6] In future, when *Ad Hoc* Integrity Constraints are implemented, these will also have to be deleted from the relevant Meta DB relvar.