Comments on
"The Architecture of the RAQUEL DBMS"

'Background' Slide
RAQUEL is actually older than **Tutorial D**, but qualifies as a valid **D**, and has been accepted as such by Chris Date and Hugh Darwen. RAQUEL was always designed to represent the relational ideas expressed in Chris Date's series of books "*An Introduction to Database Systems*", and so has naturally evolved into a valid **D**.

'Design Aims' Slide
Point 1 gives an example of a RAQUEL statement. It shows a literal relational value (= relvalue) of 3 tuples, the first of which is written out in full and consists of an attribute A with a numeric value of 1 and an attribute B with a text value of 'Jon'. The relvalue is inserted in relvar *R2*. Then relvars *R1* and *R2* are (naturally) joined together on attribute A. The result of the join is retrieved and sent to *S*, which is a Sink (say an application which receives the results of the query).

A RAQUEL statement consists of a relational assignment expression. **<--Retrieve** and **<--Insert** are assignments. Assignments carry out actions on the database. An assignment takes either one or two relational algebra expressions as operands. The **Join** expression and the relvalue are examples of algebra expressions; so too is the sink *S* because it looks like a relvar from within the database.

Point 2 concerns the fact that the RAQUEL architecture is open, not closed, and that it consists of building blocks. Together they allow a variety of RAQUEL DBMS configurations to be created.

This strategy originally came about to facilitate student projects; each project would concern one building block. It was then realised that this approach was essential to build a DBMS via the Open Source route, so that each contributor could focus on their building block and ignore the rest of the DBMS. Each building block is a 'Black Box' with defined interfaces.

'Architecture' Slide
The Logical Architecture defines what and how the DBMS functions. The Memory Architecture defines approaches that the program code must adhere to in order to eliminate inefficiency. The Physical Architecture defines how the Logical Architecture is applied in program code in a way that achieves the Memory Architecture. The Physical Architecture maps the Logical Architecture into code modules that are procedures; in effect the Logical Architecture determines a functional decomposition approach to programming. However object classes are also used to raise the level of abstraction of program code.

‘Reason for Compactor’ Slides 1 and 2
In order to have a simple Tokeniser and a simple Parser to carry out their traditional functions, it has been found useful to have a Compactor that compacts the ‘word tokens’ produced by the Tokeniser into ‘RAQUEL Tokens’ that the Parser can use.

Some operators and assignments have parameters – enclosed between ‘[‘ and ‘]’ – that are themselves expressions and hence need to be turned into parse trees. The example shows a **Left Semi Generalised Join** (note the additional asymmetric left bracket to denote an asymmetric operator) that uses a comparison expression as a parameter to drive the join. The Compactor therefore needs to apply the Tokeniser, Compactor (itself) and Parser recursively to the parameter comparison expression to turn it into a parse tree. The parameter parse tree is logically a component of the RAQUEL token that denotes the **Left Semi Generalised Join**.

The recursion can continue to any depth, and is useful for handling the relational expressions as parameters that arise from nested relations. (The example shows a scalar expression).

‘Ram Design Strategies’ Slide
The ‘Global RAM’ diagram is intended to show that when major data structures are passed between successive DBMS functional procedures (i.e. procedures that execute DBMS functions), they are not copied from the internal RAM of one procedure into the internal RAM of the next procedure. Rather the data structure stays in RAM that is local to the DBMS but global to each procedure, and a pointer to that RAM is passed from one procedure to the next.

The purpose of this is to eliminate copying between areas of RAM wherever possible. If the DBMS has to do such data copying on a large scale, it could become a significant overhead.

The diagram also shows that where the DBMS internal RAM is spread over two processors, a “DBMS RAM procedure” handles the transfer of data between the two processors so that the functional procedures do not have to handle this.