



Web-Based Cost Estimating

This project developed a Web-based cost-estimating system for a small U.S. manufacturer in the aerospace industry.

It allowed their subcontractors to quote directly on subparts contained in a central database of manufactured parts. Quotations are immediately reflected in the manufacturer's overall costing of their own product. The database containing this information can also be used to query financial information in any way, and be integrated with other database-driven activities, such as scheduling.

Traditionally, cost-estimating is done with spreadsheets on standalone PCs because manufacturers do a significant amount of "what-if" calculations while preparing quotes using vendor-supplied information. There are several problems with this approach:

- Spreadsheets can't properly represent the hierarchical nature of complex parts, which are composed of subparts which may also be standalone products
- Updating information from vendors to spreadsheet is generally cut-and-paste
- Linking spreadsheet information to other databases is difficult

Using the Web, vendors are now able to quote directly on component parts of the manufacturer's master database, updating it automatically. This information can then be subjected to database queries or spreadsheet-like operations, or linked to other databases.

Our project has solved the problem of building parts-oriented databases which act like spreadsheets, by using a tree-based data model, and replacing algebraic formulas linking cells with SQL operations performed on nodes.

Technical Description

The database model represents manufactured end products as trees, where the nodes are subparts related to each other in an hierarchical manner. User-defined attributes on each node define properties relevant to its cost, such as labour and material.

Manufacturers may interactively build any tree representing an end product, and assign to it whatever attributes deemed important. After the values of those attributes have been entered from the Web, manufacturers may then interactively build queries which perform spreadsheet-like operations on the data.

- Database Model

The database model below is based on the notion of sub-assembly (SA) and detail part (DP) which fit together to form larger sub-assemblies. The purpose of this model is to allow users to assign values to custom-defined attributes on each sub-assembly and detail part during the construction process (such as cost or type of material used) and then aggregate total values for each attribute in the final sub-assembly. Users are able to query on these attributes to break

down total values based on some criteria (such as total cost in the final sub-assembly for a given type of material).

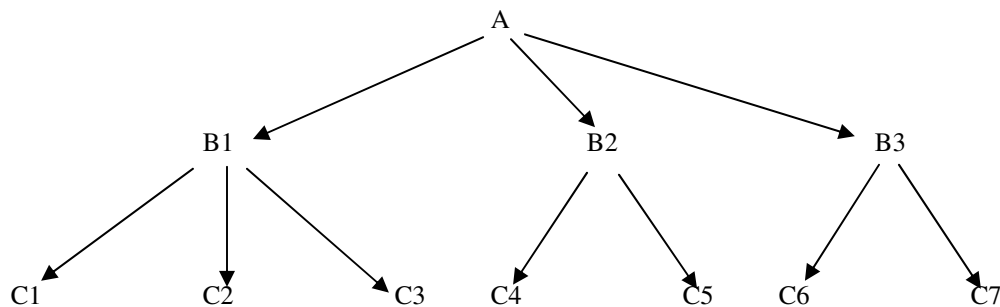
Detail parts, by definition, cannot be broken down into smaller objects.

A purchased standard (PS) is an end item manufactured by supplier. Typically this might be a bushing, bearing, rivet etc. It should be noted that purchase standards are viewed here as a special kind of detail part, namely those that are procured from authorized vendors. Otherwise, there's no difference.

The model is built on two standard notions in database design: *object-attribute-values* and *trees*.

Objects can have attributes (such as weight) whose values (such as 1 Kg) help define the object in question. Typical attributes in manufacturing processes include weight, length, width, color, type of metal, etc. Attributes can be numeric or categorical (ie. color) where, in the latter case, no arithmetic computations are possible. From a database perspective, *objects are completely defined by the values of all their attributes*.

Trees may be defined mathematically, but for the purposes of this demonstration a simple example will suffice:



Each tree has a root node (A) and various subnodes (B1, B2, B3). Each subnode may have its own subnodes, and all of them together are called the descendants of the root. The nodes at the bottom are called leaves. Furthermore, each subnode may be viewed as the root of its own tree, so that trees can be recursively defined in terms of smaller trees, called subtrees. The importance of this is that any computer program written for a tree will also work for all its subtrees.

Trees are very useful for describing hierarchical relationships, and are easy to program in software. For example, the above tree could represent a corporation (A) with its subsidiaries (B1, B2, B3), each having its own subsidiaries. The corporation A has 10 descendants.

Combining trees with attributes provides a clear and compact model for describing hierarchical relationships. For example, if the attribute called revenue is applied to each node in the above tree, then the total revenue for the root is simply the sum of that attribute on its subnodes and their successors. A computer program doesn't need to know the meaning of nodes or attributes – it only has to add the values of attributes or perform queries on them. To enhance the program later usually implies adding more attributes, or restricting the kind of tree it handles.

In this model trees represents sub-assemblies, whose attributes include such things as cost. Detail parts are always leaves in any tree, although sub-assemblies may also be leaves. For

example, a sub-assembly may be a leaf in a larger sub-assembly if its constituent parts are irrelevant to the manufacturing process for the larger given sub-assembly (eg. they're purchased elsewhere, and only its attribute values are required, such as price). Or the sub-assembly may be assembled in the manufacturing plant, but only its attribute values are required for the costing of the larger sub-assembly. If its constituent parts are later required, then the tree representing the larger sub-assembly need only be changed by replacing the leaf representing the smaller sub-assembly with its own subtree.

Each SA is composed of other SA and DP components. In the former case, further partitioning is possible, so that a nested sequence of SA and DP components yields the original SA component. For example,

$$SA0 = SA1 + SA2 + DP1$$

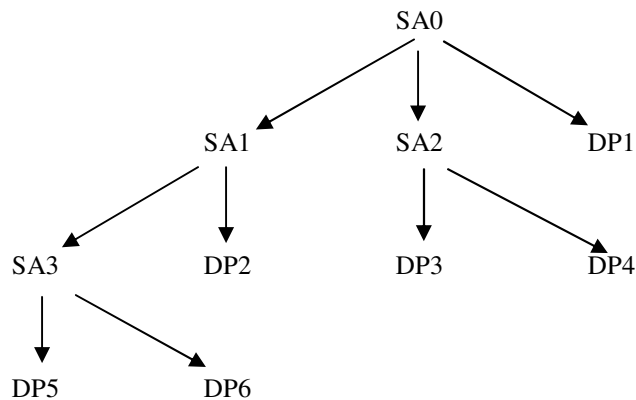
$$SA1 = SA3 + DP2$$

$$SA2 = DP3 + DP4$$

$$SA3 = DP5 + DP6$$

or

$$SA0 = [[DP5 + DP6] + DP2] + [DP3 + DP4] + DP1$$



This notation means that SA0 can be broken down into 6 DP components, built using a four-step process, each yielding an SA with its own identifying ID.

Now, to extend this example, suppose that two attributes are defined for each DP component:

Cost(DP) = cost of building component DP

Material(DP) = material used for building component DP

Values for these attributes may be maintained in lookup tables, as follows:

$$\text{Cost}(\text{DP1}) = 1$$

$$\text{Cost}(\text{DP2}) = 2$$

$$\text{Cost}(\text{DP3}) = 3$$

$$\text{Cost}(\text{DP4}) = 5$$

$$\text{Cost}(\text{DP5}) = 8$$

$$\text{Cost}(\text{DP6}) = 13$$

Material(DP1) = Aluminum
 Material(DP2) = Aluminum
 Material(DP3) = Aluminum
 Material(DP4) = Copper
 Material(DP5) = Copper
 Material(DP6) = Copper

The total cost for SA0 can then be made available by the following recursive query:

$$\begin{aligned} \text{Cost(SA0)} &= \text{Cost(SA1)} + \text{Cost(SA2)} + \text{Cost(DP1)} \\ &= [[8 + 13] + 2] + [3 + 5] + 1 \\ &= 34 \end{aligned}$$

Similarly,

$$\text{Cost(SA0, Aluminum)} = 6$$

The sub-assembly SA0 may be viewed as the end item, ie. a deliverable to the manufacturing client. However, if any of its subtrees is viewed as an end item, the same computations may be used for costing purposes.

Some of the attributes for detail parts include material and processing requirements (any finishing treatment, including heat treating, painting, plating, non-destructive testing, etc.)

- Database Programming

Trees are represented by linked tables listing nodes and their immediate successors (Fig. 1). Various Visual Basic functions are used to traverse such trees in a recursive manner.

Fig. 2 shows the interface for defining specific trees, attributes and their values.

Spreadsheets are built interactively using SQL statements, which are generated automatically by the program.

The nodes of any tree represent the rows of an SQL spreadsheet, while its (numeric) attributes represent columns (called standard columns).

Fig. 3 shows how new columns are interactively derived from the standard columns (and other previously derived columns) using algebraic operations. These new columns are stored internally as SQL statements which the program generates automatically.

Fig. 4 shows how columns are joined together to form spreadsheets, which are represented internally by the program as SQL statements, eg.

```
SELECT [@_Material_Cost3].Node, [@_Material_Cost3].Material_Cost_each,
[@_Material_Cost3].Processing_Costs_each, [@_Material_Cost3].Sub_Contracting_Costs_each,
[@_Material_Cost3].Standards_Costs_each, [@_Material_Cost3].Material_Cost,
[@_Material_Cost3].Material_Burden, [@_Material_Cost3].First_Year_Assertion,
[@_Material_Cost3].G_A, [@_Material_Cost3].Profit, [@_Material_Cost3].Total_Material,
[@_Material_Cost3].Process_Costs, [@_Material_Cost3].Sub_Contracting_Costs,
[@_Material_Cost3].Processing_and_Sub_Contracting1, [@_Material_Cost3].Material_Burden1,
[@_Material_Cost3].First_Year_Assertion1, [@_Material_Cost3].G_A1,
[@_Material_Cost3].Profit1, [@_Material_Cost3].Total_Processing_and_Sub_Contracting,
[@Standard_Costs_Sum].Standard_Costs, V(DLookUp("Value", "tblConstant", "Id =
21"))*V([Standard_Costs]) AS Material_Burden_2,
(V([Standard_Costs])+V([Material_Burden_2]))*V(DLookUp("Value", "tblConstant", "Id = 24"))
AS First_Year_Assertion_2,
(V([Standard_Costs])+V([Material_Burden_2])+V([First_Year_Assertion_2]))*V(DLookUp("Value
```

```

", "tblConstant", "Id = 26")) AS G_A_2,
(V([Standard_Costs])+V([Material_Burden_2])+V([First_Year_Assertion_2])+V([G_A_2]))*V(DLo
okUp("Value", "tblConstant", "Id = 28")) AS Profit_2,
(V([Standard_Costs])+V([Material_Burden_2])+V([First_Year_Assertion_2])+V([G_A_2])+V([Pro
fit_2])) AS Total_Standards
FROM [@Standard_Costs_Sum] INNER JOIN [@_Material_Cost3] ON [@Standard_Costs_Sum].Node =
[@_Material_Cost3].Node;

```

Because of the number and complexity of these SQL statements in the final spreadsheet for an end product, it is not possible for users to manipulate them manually.

Fig. 5 lists a simple spreadsheet.

Spreadsheets may also be joined together, and a given spreadsheet may have additional columns added interactively (Fig. 6) by defining more algebraic relationships between them.

Spreadsheet constants are defined and stored in an SQL lookup table (Fig. 7). The program automatically calls a Visual Basic lookup function whenever it encounters a constant name within an SQL statement. Users perform "what-ifs" by modify the values of those constants within that table, before re-listing the spreadsheet.

Aggregate functions, such as Sums or Standard Deviation, cannot be represented by building new columns, since they represent "vertical" operations on fixed columns, as opposed to the usual "horizontal" definition of a new column which base its value on neighbouring columns in the same row. They need to be their own spreadsheet, joined as before. Fig. 8 shows how this is done.

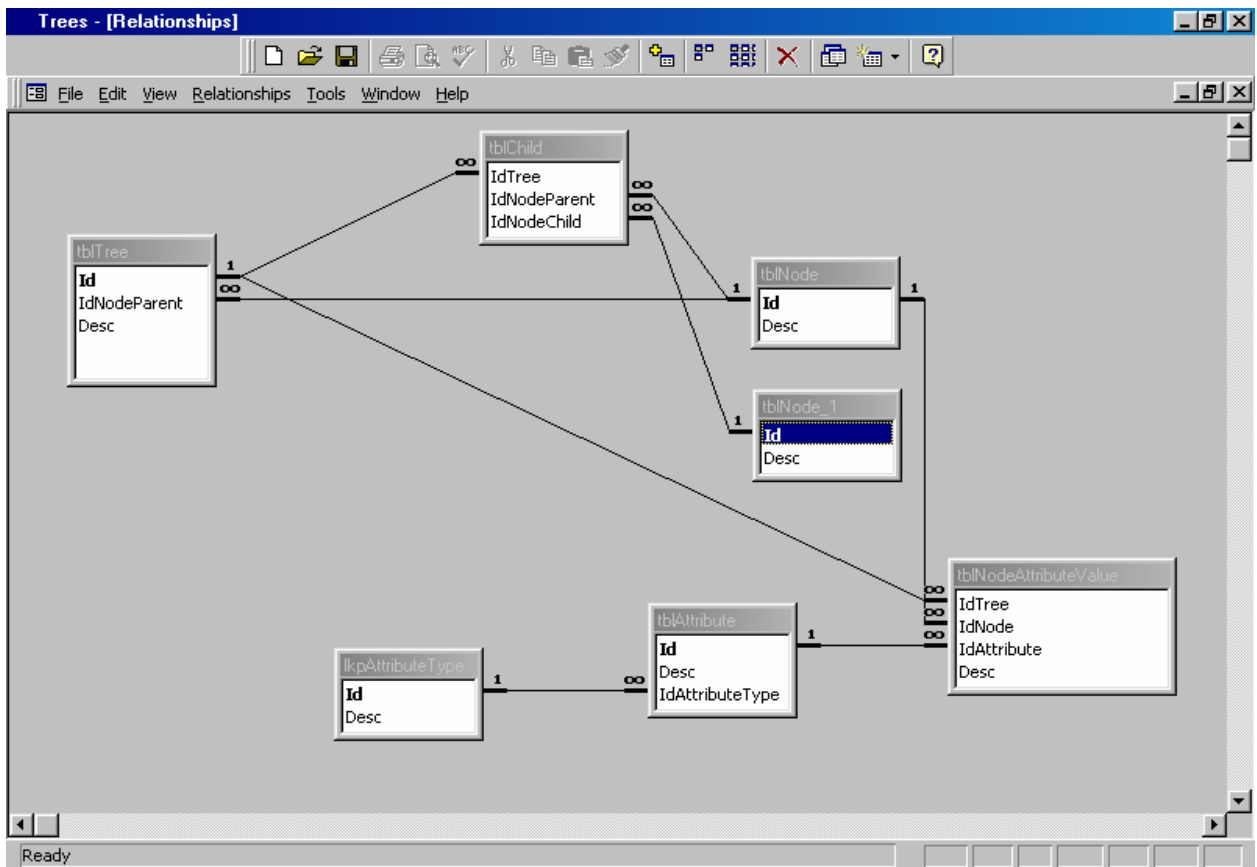


Fig. 1 Representing Trees As Linked Tables

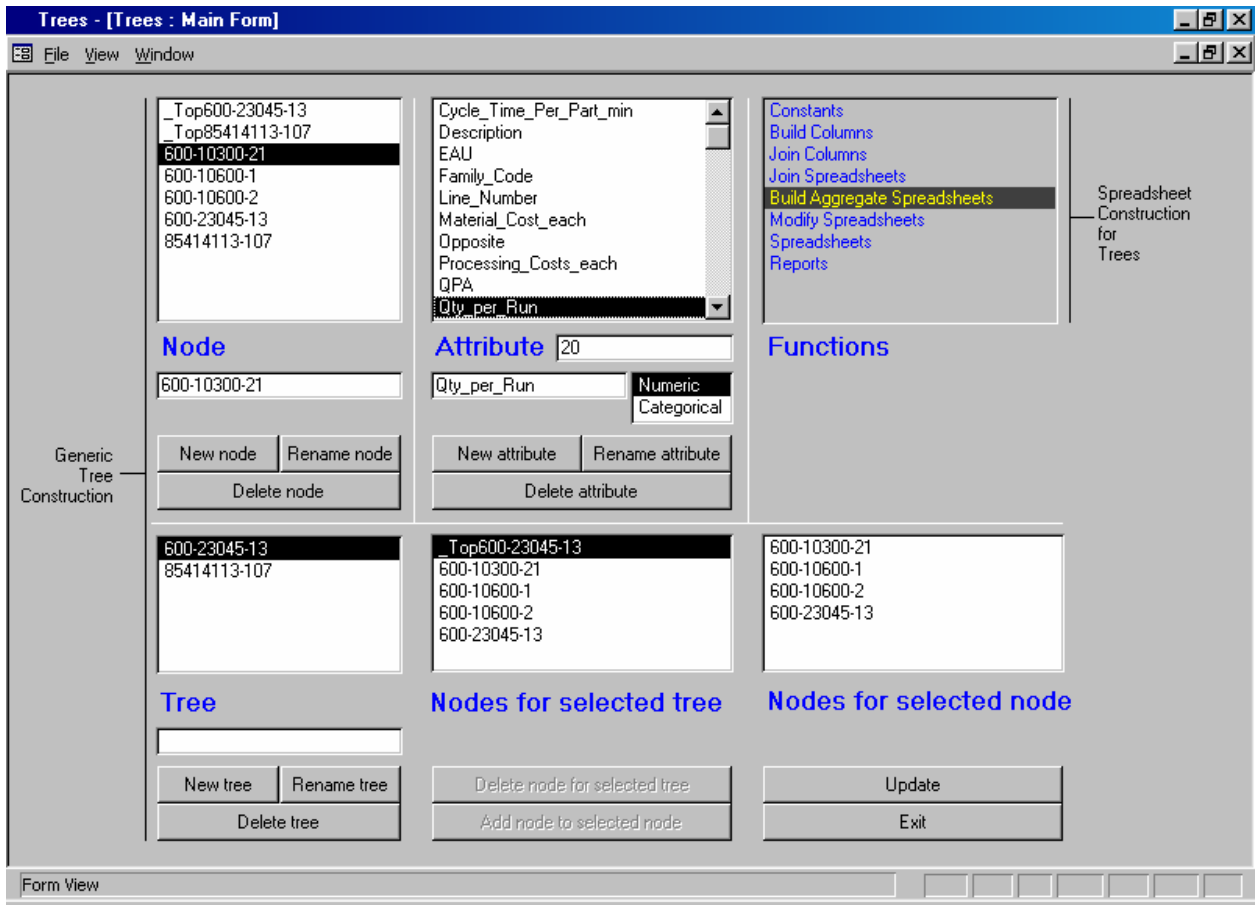


Fig. 2 Defining Trees and Attributes

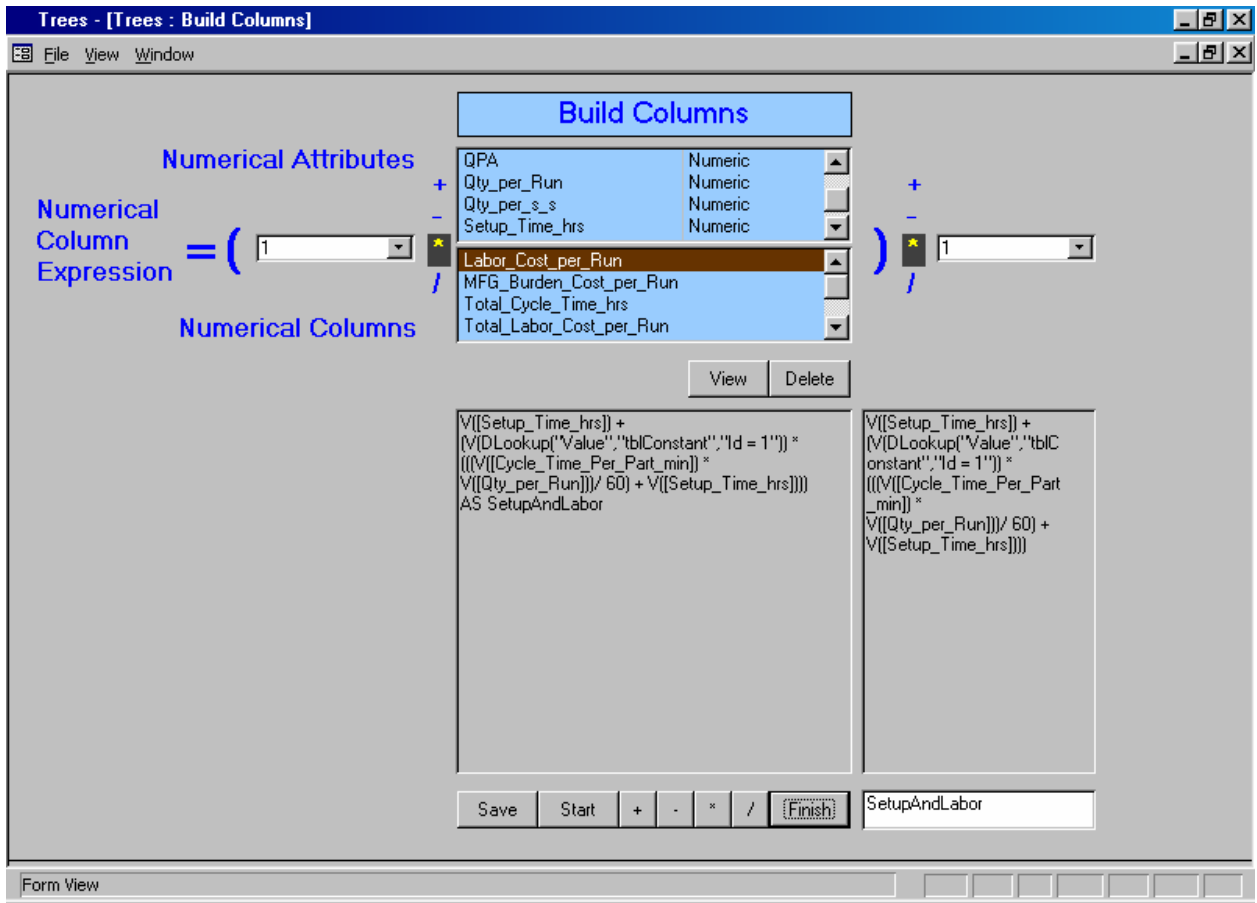


Fig. 3 Building Columns Using Attributes

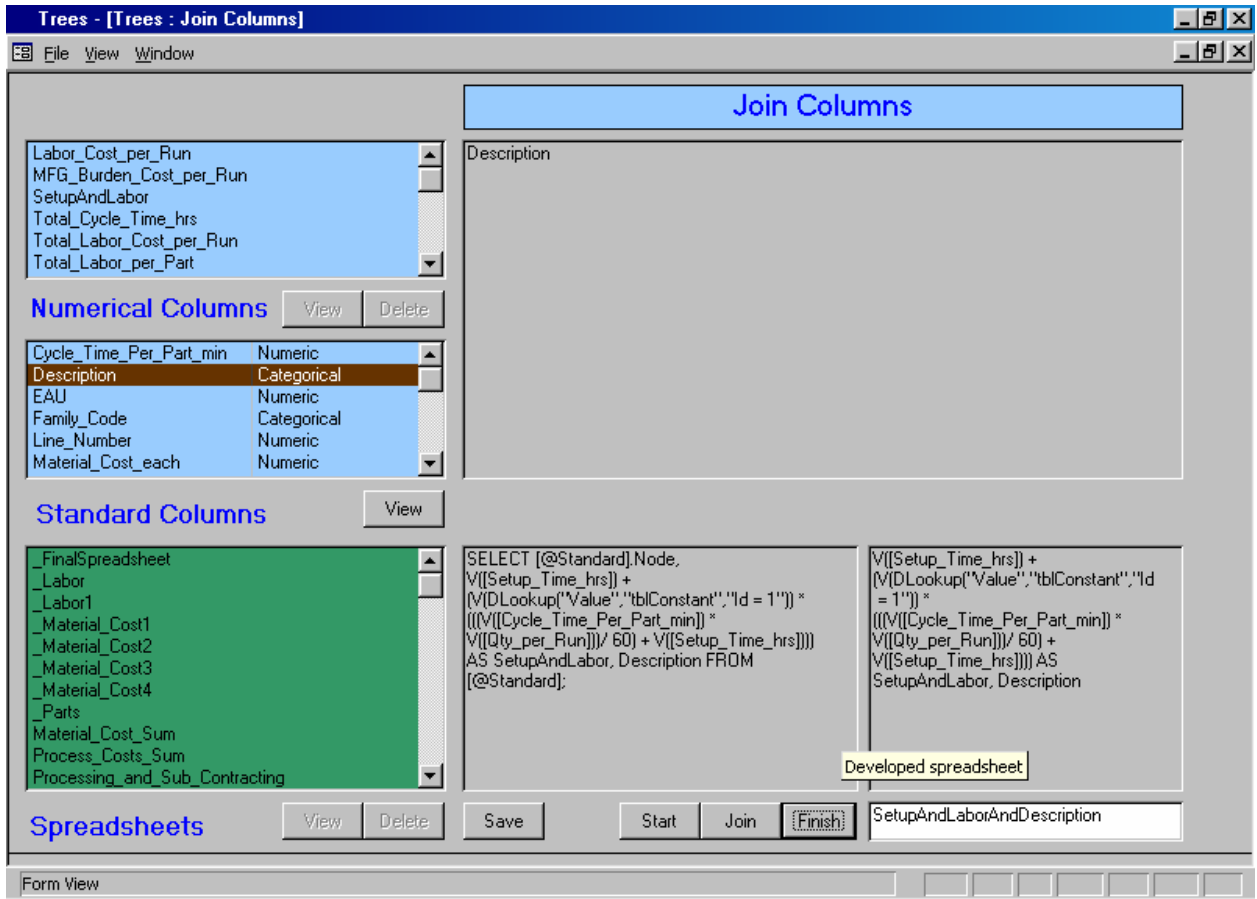


Fig. 4 Joining Columns To Build SQL Spreadsheet

Trees - [@SetupAndLaborAndDescription : Select Query]

File Edit View Insert Format Records Tools Window Help

	Node	SetupAndLabor	Description
▶	85414113-107	549.5	Fitting XN210 Outboard M/C
	600-10300-21	464	Fitting
	600-10600-1	53	Fitting Assy
	600-10600-2	545	Fitting
	600-23045-13	589	Fitting Assembly

Record: 1 of 5

Datasheet View

Fig. 5 Viewing SQL Spreadsheet

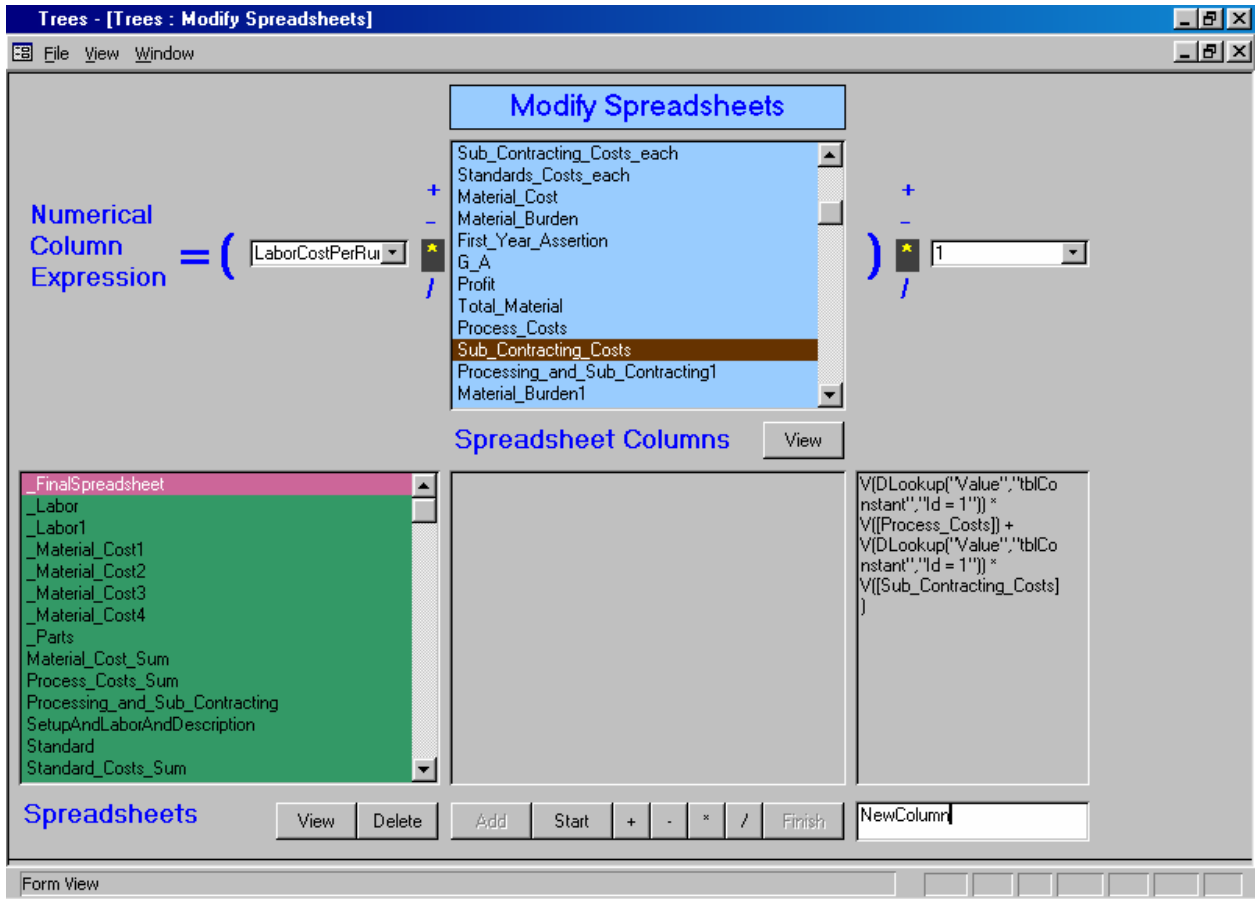


Fig. 6 Modifying Spreadsheet By Adding New Columns

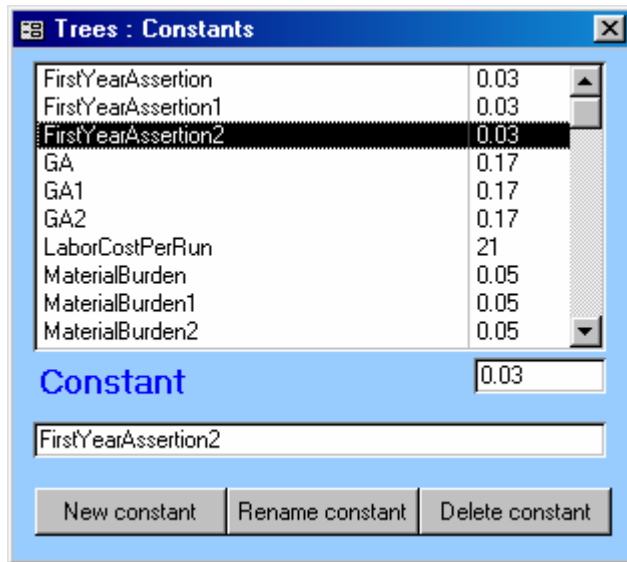


Fig. 7 Defining Spreadsheet Constants

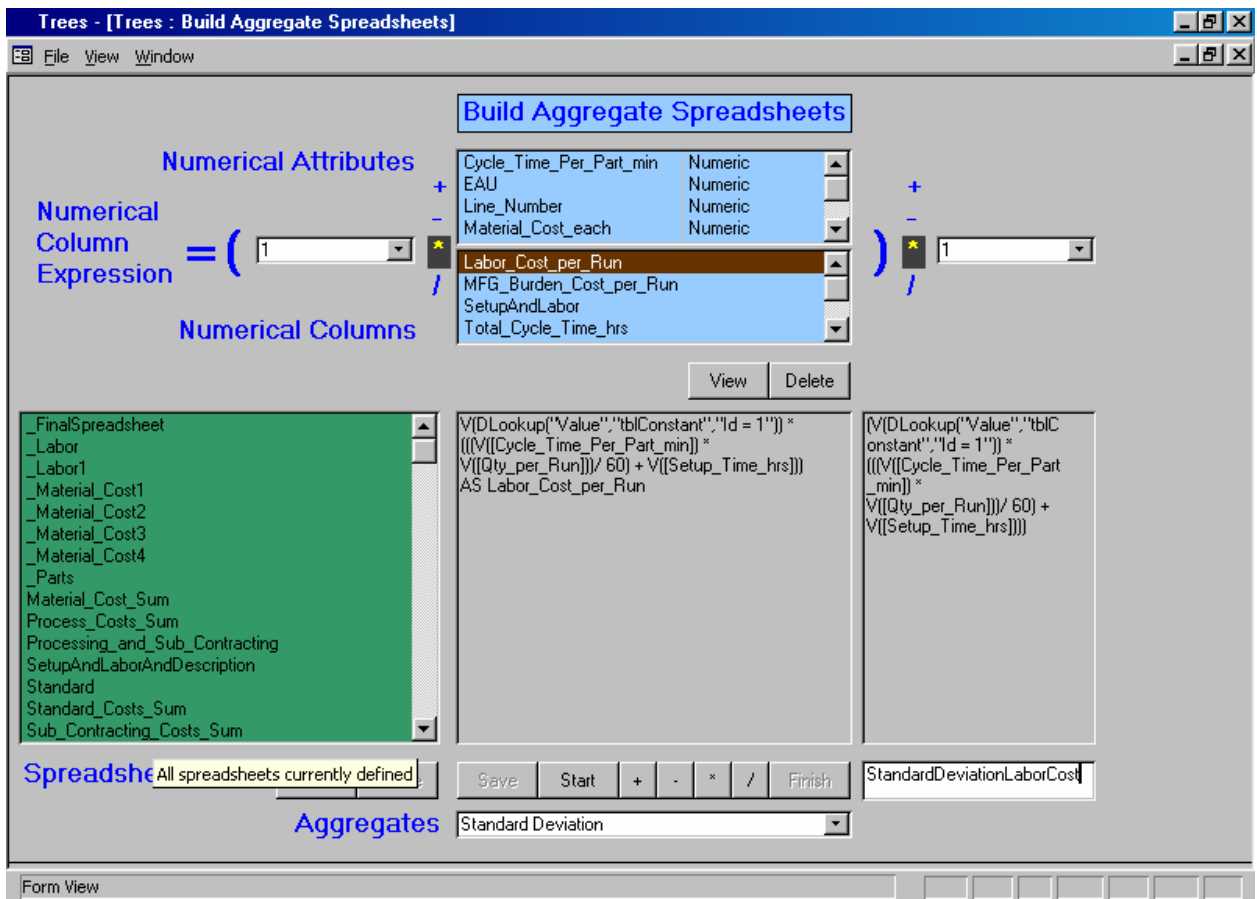


Fig. 8 Building Aggregate Spreadsheets