Presentation 1090
Beyond OLAP: The Contextual Advantage

Victor E. Hansen
MKS Inc.
992 Old Eagle School Road, Suite 906
Wayne PA 19087

Introduction

Computer software has not yet delivered what we had hoped. The computer hardware industry has been doubling price-performance every 18 months for some time and will continue to do so. No such gains have been seen in the software industry.

The year 2001 is close at hand, yet no "HAL" technology is even on the distant horizon. The World Wide Web has millions of pages of information, but finding specific pages of interest can be frustrating if not impossible.

Why is it that the software component has been so difficult, so expensive, so slow to mature? There are many reasons, and this paper will examine one critical reason which until now, has been overlooked by the software industry. In addition, this paper will introduce V4, a software tool, which addresses this issue.

We can view software as the interface between the processing power of computer hardware and the requirements of people. A great deal of effort has been expended to allow software to efficiently utilize computer hardware. The same effort needs to be made to allow software to interact with people. One way to achieve this is by creating software which emulates the behavior of people.

People have an inherent ability to incorporate the notion of context. The meanings of words change from moment to moment ("John had a ball at the ball"). Facts change with different contexts (Tim was single, then married, then divorced, then married again.). People react differently in different contexts (seeing someone fall off a building in a movie can be hilarious, not so in "real life"). People use different strategies in varying contexts (a telephone number can be dialed from memory, by pushing a speed dial button, by asking a co-worker, by calling information, or using a telephone directory).

How does this relate to programming? A fundamental understanding of context will have an enormous impact on many areas of software development and application. A few of these areas are described below.

- Code Reuse- Too much code is being custom written for each new application. Software engineers have not been able to write software in reusable, standardized pieces. Code reuse can be rephrased as "a problem of using the same code in different contexts." If software could automatically adapt to different contexts then standardized routines could be used without having to reinvent the wheel time and time again.

- Artificial Intelligence- AI encompasses many areas of research. One area is the study of natural languages such as English or French. Progress in natural language understanding has been limited to specific knowledge domains or contexts. A better understanding of how context influences the meanings and uses of words would advance this area. Software which enables robots to navigate on their own is another domain of AI. Writing programs to handle all the different situations a robot may encounter has proven extremely difficult. Again, if context could be used to determine what a robot is to "do next" then the design and implementation of these programs could be simplified.

- Expert Systems- An expert system is a program which, given a problem, interacts with the user to learn more about that problem and then offers solutions. Expert systems have been used for medical diagnosis, configuration pricing, and industrial process control, to name a few. Constructing the rules for these expert systems is not easy. Rules have to be drawn from human experts and coded into programs. These rules often overlap and are sometimes contradictory. If we could specify the context in which each of the rules applies, then the creation of expert knowledge bases could be simplified.

- The Internet- The internet currently suffers from two problems. The first is that people are having trouble finding the information they want. The second is that people are finding the (inappropriate) information they want! In order to find information, users typically access web pages through one of two ways- search engines which retrieve a list of pages based on keywords and links embedded within web pages.

  Both methods would benefit from contextual handling. Keyword searches would be able to differentiate between the possible meanings of “hot dog” (a food, a warm canine, a crazy skier). Contextual links within web pages would be able to link pages by interest, by level of education, etc.

  The second problem with the internet is restricting access to certain information. Sites one group of people may find offensive or obscene may be perfectly acceptable to another (fundamentalist religious groups and 20-year-old male fraternity members). The ability to restrict access by contextual preference would certainly make browsing the web more acceptable to many groups.

- Business Systems- A business succeeds over a long period of time because it is able to adjust continuously to changing needs and demands; to paraphrase, a successful business easily adapts to a changing context. The application software that supports a business must be able to adapt easily as well. This is usually not the case and considerable expenses are incurred to keep application software in tune with the business.

  Product pricing within an order entry system is a simple example. On what is the price of an item based? It might be based on dates, on quantities sold, on customer contracts, etc. An order entry program and supporting database must be custom designed to handle pricing. Once implemented, a simple change such as “give a two percent discount for all orders taken on weekends between Thanksgiving and Christmas” can result in costly and error-prone code/database changes.

- Personalization & Customization- Everybody has his own preferences, and many application programs recognize this by providing a way to alter fonts, tools, windows, etc. Unfortunately, this is done in an ad-hoc way which provides little consistency as users move among programs and systems. The end user would benefit immensely if an application program were more aware of the context in which it was run. User productivity could be greatly increased with a global context that all applications could access. Degrees of skill, layout preferences, user handicaps, and many other attributes could be included in the contextual base.

The above examples represent only the tip of the iceberg. A deeper understanding of how context operates and its applications to software will impact all areas of software development and use.

This paper will focus on how context can be used to model more effectively complex databases and, with this new model, to retrieve and manipulate data. First, we will define context as it relates to a computer system. Second we will explain how this relates to modeling databases. Third, we will briefly introduce V4, an implementation of this model. Following this introduction will be several relatively complex examples of how V4 can solve non-trivial problems. These were chosen over more basic examples to better differentiate this model from the many models and tools currently available.

What is Context?

The notion of context can be defined informally as those things or concepts which we are currently "dealing with" or "talking about." The hard part is trying to formally define context to a computer. If we are talking about a person, how do we represent the *essence* of that person. Or how do we represent an inventory item, or a sales territory? None of the obvious solutions work (trust me, I've tried them!). A row in the Customer Master relation does not represent a customer- it is merely a list of facts about a customer. A customer code or identifier does not represent the concept of a customer any more than the name "John Doe" represents a unique individual.

In order to have a context we need to create abstract points which represent the *essence* of something. Our context will be defined as a set of these *essence* points. As a further refinement we will collect these points into groups called **dimensions**. For instance, if we want to represent inventory items, we create an "Inventory" dimension and declare that each **point** along that dimensional line represents a unique inventory item. Similarly we create dimensions for any concept we wish to model and assign a point along that dimensional line to each instance of the concept. Examples of dimensions include-

- Integer - the points are all of the integer values (1, 2, 5943, …)
- Date - each point represents a different date (28-Aug-1997, 1-Jan-2000)
- Alpha - each point represents a different alpha-numeric string
- Module - each point represents a programming module
- RelationXYZ - each point represents a row of the relational table XYZ
- Attribute - each point represents a database attribute or field (DateOfBirth, ZipCode)

A Data Model

What do dimensions and points have to do with databases? We can model a database relation or table as a dimension and each row of the relation as a point along that dimension. Note that each point represents a row, not the contents of the row. We can create another dimension for attributes or fields, where points along this dimension represent the columns of the relation. The values stored within the relation are referenced by combining or **intersecting** a row point (a point on the relation dimension) and a column point (a point on the attribute dimension).

Now we are ready to make a conceptual leap! Instead of modeling an inventory master relation, let's model the actual inventory items. Instead of having a relation dimension and points (rows) of the relation, we will have an inventory dimension with each point on the dimension representing a different inventory item. This may seem to be identical to the first model but it differs in several important ways:

- We are no longer tied to a particular database scheme. We obtain information about items by intersecting a point on the inventory dimension with an attribute point. We can extend this to customers, vendors, employees, orders, etc. without having to know the specific tables or database schema.
- We can now "link" data with conceptual points instead of the traditional "Id's" (keys) of databases. As an example, an order references a customer. In the relational model, the order master would typically contain the Id of the customer being referenced. In our new dimensional model, an order (represented by a point!) would intersect with the "Customer" attribute point and the intersection would be a point on the customer dimension.

- This dimensional model is in no way limited to just two dimensions (rows and columns). In fact many aspects of the real world require far more than two dimensions to be modeled adequately. The price of an inventory item is a good example. The price of an item can vary with time, by customer, by contract, by catalogue, or by quantity ordered. Sales Analysis history is another good example - it can vary by any number of dimensions, including time, salesrep, customer, state, territory, line-of-business…
- Dimensions can be defined which *describe* the data we are modeling. This is know as **metadata**. We can use metadata to define upper and lower limits to values, default values, and descriptions. It is even possible to describe the organization and interrelationships of the data and to automatically construct queries based on general row-column requirements.

What About Context?

In order to represent context we created abstract dimensions and points which we then used to model databases via two-dimensional (row-column) intersections and then higher dimensional constructs. What happened to context? We can use context figuratively to “fill in the blanks” when we have complex, multi-dimensional data.

The price of an inventory item is a good example. As mentioned above, the price of an item may depend on many different parameters or components. The price of item A may be fixed, the price of item B may change on a particular date, item C’s price may vary by the number being purchased, and D’s price may be based on quantity and customer. If we did not know ahead of time what determined each item’s price, we would have to specify all possible conditions associated with pricing every time we needed the price of an item. A similar situation would be having to give the full name of a person every time that person was mentioned in a conversation. We seldom use full names, instead we use just first names, and sometimes only “he” or “she,” to refer to a person. We can get away with this abbreviated form because the conversational context supplies the additional information. We can do the same thing with data references- ask for “the price of item x” and use the current context to supply the date, the customer, the contract, the quantity, or whatever else is needed to calculate the price. This gives us great flexibility in implementing systems in that we do not need to explicitly reference each data element; we can use the conversational equivalents of first names or pronouns and let the runtime context fill in the details.

Using context gives us another capability- what-ifs. If we can use context to determine a specific reference then we should be able to create “what-if scenario” contexts. Different contexts can be defined to change the values and even the structures of existing data. The original values and structure do not actually change, they only appear to change- in the context of the what-if scenario. This type of power is unique to the context-sensitive model. What-ifs can be used to re-analyze history (What if the price of item A had been increased five percent for customer Z?). What-ifs can also be used to examine different possible futures (What if flight hours increased ten percent next spring?)

V4 - An Implementation of this Model

The V4 model is a multi-dimensional database. The basic components of the model are dimensions which represent concepts (numbers, dates, customers, inventory items, procedures). Points are instances along a dimensional line (34 is a point on the integer dimension, 11-Nov-96 is a point on the date dimension). Values are stored at the intersecting points within the multi-dimensional cube.

The syntax for a point is *dim:point,* where *dim* is the name of the dimension and *point* represents a point along that dimensional line. Examples of V4 points are:

Date:961102	
Alpha:“A text string”	*(can also be notated as simply “A text string”)*
Int:1234	*(also just 1234)*
List:(1234 “A string” Date:961102)	*(a point which is a list of other points)*
Address	*(a name with no dimension is on the Name dimension)*

Series of points and ranges of points can be specified where they make sense:

Month:9601..9612	*(the months Jan-96 thru Dec-96)*
Int:1234,1236,1239	*(the integers 1234, 1236, 1239)*
DaysOfWeek:Mon,Tue,Fri..Sun	*(Monday, Tuesday, Friday, Saturday, and Sunday)*

Finally special references can be made:

Cus*	*(represents the current customer)*
Sales:>10000	*(represents sales greater than 10000)*
Vendor..	*(all points on the Vendor dimension, i.e. all vendors)*
Intersections are notated by enclosing points in brackets-	
[Name Cus*]	*(represents the name of the current customer)*
Cus.Name	*(is equivalent to [Cus* Name])*
[Price Item:xx Date:961113 Cus:XYZ]	*(the price of xx on 11-Nov-96 for customer xyz)*

Modules, or procedures, are also considered points and can be referenced anywhere. Modules return points as values allowing nesting. Recursion is also supported. Many modules support optional arguments. These are called "tagged" arguments because they must be tagged with an argument name (*tag::value*). Tagged arguments may be specified in any order. Examples of modules are:

Plus(Int* 123)	*(returns the sum of the current point on Int dimension plus 123))*
DTInfo(Date* Year?)	*(Date-Time info- returns the year corresponding to current date))*
Sort(Cus.. By::Cus.Id)	*(returns list of customers sorted by Id., By::Cus.Id is a tagged argument))*

Values are associated with the intersection of points with a Bind command. Values may be literal points, procedures, or references to other intersections:

```
Bind [Name Cus:xyz] "The Smith Company"
Bind [Price Item:1234 Date:961101..961231] USDollars:12.99
Bind [Month Order..]                        (The month of any order is via procedure DTInfo())
   DTInfo(Order.EntryDate Month?)
Bind [CreditLimit Order..]                  (The credit limit of any order
   Order.BillTo.CreditLimit                  is via the credit limit for the billto customer)
Bind [TaxCode Order..]                      (The tax code of any order
   Order.ShipTo.TaxCode                       is via the tax code for the shipto customer)
```

V4 has powerful primitives for iterating or enumerating through points in a list or dimension. For example to create a table of all customers with 1995 sales greater than 150,000 sorted by state and customer code one could use the command:

```
Enum(Sort(Cus.. If::GT([Year:1995 Sales] 150000) By::Cus.State By::Cus.Id)
   @EchoT(Cus.Id Cus.Name Cus.State Cus.Rep.Mgr.Name))
```

which shows the customer id, name, state, and the name of the manager of the sales-rep for that customer. Notice that the intersection [Year:1995 Sales] did not reference the customer. This is because V4 "knows" that we are referring to customers, and automatically pulls sales for the correct customer. The point "Cus.Rep.Mgr.Name" demonstrates how V4 can link different pieces of information. By reading the point backwards we determine that it references "the name of the manager of the sales rep for the current customer". Another way of generating the same table is with the Matrix routine

```
Matrix(  Rows::Sort(Cus.. If::GT([Year:1995 Sales] 150000) By::Cus.State By::Cus.Id)
         Columns::(Cus.Id Cus.Name Cus.State Cus.Rep.Mgr.Name))
```

The Matrix() module has many options for generating two-dimensional matrices (spreadsheets) with column and row headers, totals, and multiple values.

Another powerful primitive within V4 is the Tally module which quickly calculates aggregate figures. For example, we might determine annual sales for the previous example from sales order data for that year. This would be done with

```
Tally(Order.. (Sum:Order.Amount By:(:Order.BillTo Order.Year) Bind:[Sales])
```

which creates bindings intersecting Sales with Years and Customers to the total sales for each combination of year and customer. The bindings would be of the form

```
Bind [Sales Year:199n Cus:xxx] USDollar:nnnn
```

Example- Sales Analysis

This first example of V4 outputs a customer ranking report. The report is actually two reports joined together. The left side shows the customers ranked by sales in 1994. The first customer is the top ranking for 1994, the second customer is the second ranking, etc. The right side of the report is similar, ranking customers by sales in 1995. The third row of the report shows the third ranking customer for 1994 and the third ranking customer for 1995, which could be different customers. The columns in the report are:

- Customer’s rank in 1995
- Customer’s name (ranking in 1994)
- Customer’s Sales in 1994
- Current Ranking
- Customer’s Name (ranking in 1995)
- Customer’s 1995 sales
- Customer’s ranking in 1994

The four V4 steps to generate the report are listed below-

```
Bind [RevSales Year..] Sort(Cus.. Reverse::Cus.YearlySales)
Bind [Cus Pos.. Year..] List([RevSales] Nth::Pos*)
Bind [Rank Cus.. Year..] List([RevSales] Position::Cus*)
Enum(Pos:1..20
        Echo(
           EvalL(Context::[Cus Pos* Year:1994] ([Rank Year:1995] [Name] [YearlySales Year:1994])
           Pos*
           EvalL(Context::[Cus Pos* Year:1995] ([Name] [YearlySales Year:1995] [Rank Year:1994]))
```

The first binding, [RevSales Year..], defines a list of customers sorted in reverse order by sales for the specified year. The second binding returns the customer at a particular position in the list (for example [Cus Pos:3 Year:1994] results in the third ranked customer for 1994). The third binding returns a customer’s position (ranking) for a particular year (for example, [Rank Cus:1234 Year:1995] returns customer 1234’s ranking by annual sales in 1995).

The Enum() statement enumerates (iterates) through positions 1 through 20. For each position, it determines the customer ranked in that position for 1994 ([Cus Pos* Year:1994]) and for that customer outputs his 1995 rank ([Rank Year:1995]), name, and yearly sales for 1994. The position number itself is then output (Pos*). The customer ranked for 1995 is then output: name, 1995 sales, and 1994 rank.

The syntax of V4 may appear somewhat foreign, but what should be obvious is its expressive power. We have taken an extremely difficult report, by current database standards, and “boiled” it down to four rather simple V4 steps.

Example - Aircraft Parts Overhaul Schedule

The example in this case study outputs a month-by-month schedule showing the quantity and cost of aircraft parts due for overhaul. The example is intended for aircraft parts distributors whose clients include commercial airlines. An example of the generated output is shown below-

	Jan-97	Feb-97	…	Nov-97	Dec-97
PartABC	*qty/cost*	*qty/cost*	…	*qty/cost*	*qty/cost*
PartDEF	*qty/cost*	*qty/cost*	…	*qty/cost*	*qty/cost*
PartGHI	*qty/cost*	*qty/cost*	…	*qty/cost*	*qty/cost*
…	…	…	…	…	…

The steps to generate the matrix are the following-

1. For all components, determine the number of hours of usage, by month, based on a combination of actual hours of usage (if known) and expected hours usage.
2. Determine the month the component will require overhaul by adding up hours of usage until the maximum allowed for the component is reached.
3. Output a matrix of components showing quantity and extended cost by month.

There are several "databases" involved in this study. Although V4 does not operate on databases, we will nevertheless describe the data as if they were relations for the sake of clarity. The data to be used is-

```
Plane( SN, HoursMon, Carrier )
IM( Id, Desc, Cost, HoursOH)
Comp( Plane, IM, Installed)
```

Each plane is a point on the Plane dimension. Data available about each plane include the plane's serial number (SN), estimated hours of flight per month (HoursMon), and the carrier owning/leasing the plane (Carrier). Bindings describing this data would be:

```
Bind [Plane:xxx SN] Serialnumber
Bind [Plane:xxx HoursMon] HoursPerMonth
Bind [Plane:xxx Carrier] AirlineOwningPlane
Usage( Plane, Month, HoursUse)
```

In addition to estimated hours of usage we have actual hours of usage by the plane for months in which the plane was in the air. Bindings would be:

```
Bind [Plane:xxx Month:xxx HoursUse] HoursOfUse
```

The points on the IM dimension are inventory items. Data available for each item include the item's item code (Id), description (Desc), cost (Cost), and number of hours between overhaul (HoursOH).

The points on the component dimension represent items associated to and installed on a plane. The data include the plane (Plane), the inventory item (IM), and the date the component was installed (Installed). Note that with components and usage, "Plane" corresponds to a point on the Plane dimension, "IM" corresponds to a point on the IM dimension.

The actual V4 rules required to generate the overhaul report are shown below without comments. A detailed description follows after the rules themselves.

```
Bind [HoursUse Comp.. Month..]
  [HoursUse Comp.Plane Month*],[HoursMon Comp.Plane]

Bind [MonthOH Comp..]
  EnumEU(List(DTInfo(Comp.Installed Umonth?) Number::1000) Sum::Comp.HoursUse GE::[HoursOH Comp.IM] )

Bind [OHSchedule Month..]
  DoI{
         Tally( Comp..
                  (Count::1 By::(Comp.IM .MonthOH) Bind::[NumOH])
                  (By::Comp.IM ByList::[OHIMList]) )
          EchoS( " " " " EnumCL(Month* (Month* " " ) ))
          EchoS( " " " " EnumCL(Month* ("Count" "$Cost") ))
          Enum( Sort( [OHIMList] By::IM.Id )
                        @EchoS( IM.Id .Desc EnumCL(Month* ([NumOH],0 Mult([NumOH],0 IM.Cost) }
```

The first binding defines the hours of usage for a component in a month. The expressions “Comp..” and “Month..” correspond to “all components” and “all months”. The binding or rule states that the hours of use are based on the hours of use of the plane for the component (Comp.Plane) in the specified month (Month*). If that is unknown or undefined then assume that the hours are the default hours for the plane ([HoursMon Comp.Plane]).

The second binding determines the month of overhaul for any component. This is calculated with the “EnumEU” module, which finds the first month (from the month the component was installed) when the sum of the hours of usage for the component exceeded the hours-to-overhaul for the item corresponding to the component. The EnumEU() module may appear to be a rather specialized piece of code, but it is not, and is used to solve a wide variety of problems. Its general form is

EnumEU(*list variable limit*)

The module enumerates through points in *list*. For each point, it evaluates *variable* and holds the sum (in this case) in an internal accumulator. When the accumulator reaches *limit*, EnumEU stops and returns the current list element.

The last binding is composed of four separate actions. The first action is a Tally which iterates through all of the components “Comp..” and tallies a count by item and month of overhaul. It also tallies a list of all of the items appearing as components, and makes a list of those items.

The next two actions are “EchoS” modules which echo their arguments to a spreadsheet. The first EchoS echos a list of (month space month space...) for each of the selected months. The second echos a list of (“Count” “$Cost” ...) for each of the selected months.

The final action is an Enum() module. This module enumerates through all of the items pulled out of the components ([OHIMList]), and, for each item, outputs the following: the item’s part number (IM.Id), its description (IM.Desc) and the count and cost corresponding to the quantity of that item to be overhauled in the specified month. In the event that the above Tally() determined that an item had no overhaul quantity for a month, we want to show a zero ([NumOH],0).

Evaluating the last rule with

```
[OHSchedule Month:9601..9606]
```

generates a spreadsheet showing all components and the anticipated quantity and cost of parts due for overhaul for the first six months of 1996. It is important to note that we are not limited to a single, consecutive range of months. Other possible evaluations include

```
[OHSchedule Month:9601,9501,9602,9502,9603,9503]
[OHSchedule Month:9501..9506,9601..9606]
```

which compare Jan-96 to Jan-95, Feb-96 to Feb-95, etc. and in the first six months of 1995 to the first six months of 1996.

What if carrier XYZ informed this parts distributor of a potential five percent increase in flight hours beginning in March 1997? How could we see the effects of this increase? Very easily, as shown below.

```
Bind [WhatIf:FivePC Month:>=9703 HoursMon Plane..]
    Mult(EQ(Plane.Carrier Carrier:XYZ) [-] 1.05 1)
```

Instead of modifying any of the existing rules, we have added a new rule which states that the HoursMon of any plane (Plane..) on or after Mar-97 (Month:>=9703) assuming a five percent increase (WhatIf:FivePC) is defined to be whatever the hours per month normally would have been ([-]) times 1.05 if the carrier is XYZ. The data will remain unchanged if not XYZ. Evaluating the report with this assumption, or in the context of (notated as "| WhatIf:FivePC")

```
[OHSchedule Month:9701..9712 | WhatIf:FivePC]
```

would give a new set of numbers based on the projected five percent increase in flight hours.

This example of a "what-if" demonstrates three powerful features of V4 not found in any other OLAP model. The first is the ability to define contexts in which rules/data vary from the norm. The second is the ability to incorporate differing contexts into existing world models without modifying the model. The third, a difficult concept to label, is being able to reference the rules/data of the normal world model in the context of the "what-if". In this example, we were able to multiply the normal Hours/Month (with [-]) without knowing anything about what the normal hours were or how they were retrieved.

Example - Forecasting Raw Materials

This example shows how this model can be used to generate a forecast budget for raw materials usage by month for the year 1997. The output is a matrix with columns for each of the months and rows for each classification of raw material (commodity code). The elements of the matrix are the forecasted dollar amounts. The matrix looks something like-

	Jan-97	Feb-97	…	Nov-97	Dec-97
Packaging	123,339	123,908	…	130,000	129989
Wires	50,002	51,399	…	55,219	54,389
Capacitors	93,989	91,002	…	90,221	93,220
…	…	…	…	…	…

The steps to generate the matrix:

1. Based on sales history taken from 1994 through 1996, project sale units of finished goods for 1997 using the least squares linear regression method.
2. For each projected item/month explode out the raw materials required by going through the bill-of-materials file which lists the raw materials needed for each finished good
3. Convert the units required to dollars by extending the projected units by the standard cost for each raw material. If a cost adjustment exists for an item then utilize it.
4. Aggregate all raw material costs by commodity code and output a table showing months as columns and commodity codes as rows.

The databases we have are the inventory master (IM), Sales History (Sales), Bill-of-Materials (BOM), Materials Parts Explosion (BOMX), and Cost Adjustments (Cadj) as shown below-

```
IM( Id, IsFG, CC, StdCost, LeadTime )
Sales( IM, Month, Units, $Sales )
BOM( IM, Yield)
BOMX( IM, RMId, Qty )
Cadj( CC, Month, StdAdj )
```

Attributes associated with inventory points (IM) are the Id, whether or not the item is a finished good (IsFG), the commodity code (CC), the standard cost (StdCost), and lead time for ordering (LeadTime). Sales history is recorded by month in Units and $Sales. The bill-of-materials information includes yield percentage (Yield) and the component parts explosions (RMId is the raw material component and Qty is the quantity required). The cost adjustments are by commodity code and month and reflect expected percentage changes to future commodity prices.

The rules to perform the analysis and generate the matrix are shown below. Note that only four rules are required to perform this complex retrieval and analysis.

```
Bind [RMProjection Month..]
 Do(        [FcstFGSales] [SumByComCode]
            MatrixT(Rows::CC.. RowCap::CC.Desc Columns::Month* ColCap::Month* Values::[RMProj],0))
Bind [FcstFGSales Month..]
 Enum(      IM.. If::IM.IsFG
            @StatProj(Enum::Month:9401..9612 Y::[Units],0 X::Month* Method::SLLS
                        At::Month* Context::Dim:Qty Do::[Explode]))
Bind [Explode IM.. Month.. Qty..]
  Def(      IM.BOM
            Enum(IM.BOM @[Explode BOM.IM Mult(Div(BOM.Qty IM.Yield) Qty*)])
            Do(Context(MakeP(Dim:Exp)) BindQ(Exp.IM IM*) BindQ(Exp.Qty Qty*) BindQ(Exp.Month Month*)))
Bind [SumByComCode]
 Tally(     Exp.. Context::Exp.IM
            (Sum::Mult(Exp.Qty Mult(IM.StdCost IM.CC.StdAdj,M:1.0)) By::(IM.CC Exp.Month) Bind::[RMProj]))
```

The first rule invokes rules 2 and 4 which perform the forecast, explosion, costing, and aggregation. The MatrixT call outputs a matrix with the commodity codes as rows (Rows::CC..), the row captions are the commodity descriptions, the columns are the months to be projected (Columns::Months*), and the values in the matrix are the projected costs at each commodity code-month ([RMProj],0)

The second rule performs the sales projection. It iterates, or enumerates (Enum), through all of the inventory items (IM..) marked as finished goods (If::IM.IsFG). For each of these items it projects future sales with the StatProj module. This module loops through the historical months (Enum::Month:9401..9612) and calculates the "Y" values as the units sold ([Units]) for the current item and month. The "X" values are the months of history (X::Month*). The projection method is straight line-least squares (Method::SLLS). Projected values are to be calculated and, for each projection, explode ([Explode]) the units into raw materials.

The third rule performs the bill-of-materials explosion. For each item-month-quantity, it tests to see if a bill-of-materials exists (Def(IM.BOM)). If so, it iterates through each raw material in the bill-of-materials (Enum(IM.BOM)) and performs a nested explosion on the adjusted quantity. If the item does not have a bill-of-materials then an entry is made for that item recording the item code, quantity, and month. Note that this explosion can handle multi-level parts explosions.

The fourth rule sums (Tally) the raw material explosion detail by commodity code and month (By::(IM.CC Exp.Month)) and sums up the cost by multiplying the quantity by the standard cost. If an adjust factor exists for the commodity code/month then it is taken into account.

What if you had to take purchasing lead time into account for the budget, meaning the cost is to appear in the month the raw material must be ordered? This can be done with V4 without making any changes to the above commands simply by including one additional rule

```
Bind [Exp.. UMonth Factor:LT] Minus([-] Exp.IM.LeadTime)
```

This rule simply states that the month associated with each explosion detail entry is to be that month minus the lead time, in months, for that item.

How Does this Model Differ from OLAP

V4 is a true multi-dimensional model. It is not limited to a single concept cube as are many other OLAP models. Summary and detail data can co-exist within V4. General Ledger, Accounts Payable, Order Entry, Sales History, Customer Master, and any other data can be entered into V4, giving the users access not only to aggregate summary information but also to detailed information that was summarized into the aggregate, and supporting data that may be useful in explaining where the aggregate numbers originated.

We briefly mentioned the ability to reference the current value of a dimension. V4 implements a runtime context which can be used

- to reference the current customer, the current order, or the current whatever within the evaluation of an intersection.
- to evaluate an incomplete intersection, and have V4 automatically pull pertinent information from the context. For example, V4 can evaluate the price of an item within the context of an order and its associated context.
- to alter the meaning of a reference external to its use by changing the context. For example Year:1995 could refer to a calendar year for sales analysis, a fiscal year for financial analysis.
- to examine multiple “what-if” scenarios by changing rules and facts. V4 can maintain and manipulate contradictory data just like humans (“If horses can’t fly, then what was Pegasus?”)

Metadata is data about data. Current relational and OLAP technologies have a great deal of difficulty with this issue. V4 does not. Default values, help messages, and inter-relational data can all be maintained within V4 using the same model and form that is used to represent the main data. If taken to the limit, then V4 is able to utilize data about its own model to construct automatically programs for accessing the data.

Conclusions

- **Context will change the way people interact with software**. The use of context in a multi-dimensional framework allows us to have references to data which automatically adapt to the current situation. Inventory pricing based on item, date, customer, contract, catalogue, and/or quantity is a good example. Context also permits us to create what-if scenarios and test models without having to change the core of the model.
- **The multi-dimensional data model is fundamentally different from existing data models**. The use of multiple dimensions allows us to incorporate data, multiple levels of aggregation, procedures, and metadata into a single conceptual framework.
- **Metadata is necessary to guide users through complex data**. The V4 model permits us to store data about the data in the same logical/physical encapsulation. Metadata can be used to tell the user what is available and how to access it. More important, though, is that metadata can be used to construct queries for the user.
- **Two dimensions are not powerful enough for the real world**. The real world is complex and multi-dimensional. Manipulating data with two-dimensional tools (i.e. the relational model & SQL) is proving just too difficult. OLAP is attempting to bridge the gap but it is not a true multi-dimensional/multi-application tool.
- **Context is more than data**. The contextual multi-dimensional model has applicability beyond databases. Expert systems, artificial intelligence, and systems engineering are just the proverbial tip of the iceberg for contextual systems.