The Evolutionary Data Warehouse--An Object-Oriented Approach

Amy Turske McNee, Trilogy Consulting Corporation, Kalamazoo, Michigan

ABSTRACT

This paper describes techniques for designing both the front and back end of a data warehouse in such a way that companies can continue to evolve their warehouse and query tools as their business changes, instead of continuously having to restructure and rewrite their existing tools. Object-oriented design and pattern languages are also discussed.

INTRODUCTION

One of the hottest topics in the industry today is data warehousing and on-line analytical processing (OLAP). Although, data warehousing has been around in some form or another since the inception of data storage, people were never able to exploit the information that was wastefully sitting on a tape somewhere in a back room. Today, however, technology has advanced to a point to make access to this information an interactive reality. Organizations across the country and around the world are seeking expertise in this exploding field of data organization and manipulation. It is not a surprise, really, that business users want to get a better look at their data. Today, business opportunities measure in days, instead of months or years, and the more information empowering an entrepreneur or other business person, the better the chances of beating a competitor to the punch with a new product or service.

The interesting thing about this new genre of computer ingenuity, is its dynamic nature. Business drives data warehousing, and the only constant in business is evolution--or elimination. Therefore, warehouses and OLAP tools also need to be capable of change. Both need to evolve as the business evolves. Unfortunately, repeatedly, as we have spoken with those who have built "data warehouses" and OLAP tools, this has not been happening in the real world. Many businesses are on their second try to a build a successful warehouse and the tools they need to reach their data.

Who's On First?

Often, developers are trying to mold the business around the tool instead of molding the tool to meet business needs. This happens when those charged with building the tools do not understand the data, the users, or the business itself.

Many answers to design questions are found in the data. Warehouse designers need to understand the fundamental business issues stored on those tapes and hard drives. To build a successful data warehouse, the designers must understand the business and data generated from that business. If they don't, the project will inevitably fail. They also need to understand alternative design solutions to provide the most effective solution. Sometimes designers fall into the category of "If you have a hammer, everything looks like a nail." All businesses are not alike, therefore not all warehouses are going to be alike.

Many answers to questions about the user interfaces are found with the users. End users need to be more involved in interface design. They need to see work in-progress and change it to meet their needs. This will also save time, money, and energy by eliminating months of development that does not address the issues that are important to the users.

Unfortunately, something inevitably changes the moment everything is understood and delivered If the software and data storage mechanisms are not equipped to handle it, the users are back at square one. The SAS System recognizes this integral business need to evolve by providing several tools including SAS/AF® with its object-oriented capabilities, SAS/CONNECT® with its ability to access several data sources, as well as the core of the SAS System with its portability across many operating systems. With the addition of the new SAS Warehouse Administrator®, the SAS System provides and end-to-end data warehousing solution.

"The Back End"

One interesting phenomenon that has occurred with data warehousing projects is an inherent misunderstanding of what is this thing we call a data warehouse. First, what it is not. A data warehouse is not a shrink-wrapped product. Contrary to what some vendors say, you cannot buy a data warehouse. You **can** buy a product that facilitates **building** a data warehouse, or a product that is a data storage system for a data warehouse, but you cannot buy a data warehouse. Also, a data warehouse is not drill-down. Drill-down is a feature that a data warehouse provides nicely for, but drill-down occurs on the "front-end."

Okay, so what is a data warehouse. My definition: *A data warehouse is a highly organized process by which you cleanse, transform, replicate, and organize your data for fast, consistent, reliable access by a user either interactively or in batch.*

A data warehouse is a mechanism for data storage and data retrieval. Data can be stored and retrieved with a multidimensional structure--hypercube or relational, a star schema structure or several other data storage techniques. In the interest of space, I am going to leave out the discussion of cleansing, transformation, replication, and meta-data, however these are also important issues that need to be addressed and implemented in your data warehouse to ensure a success.

The Basics

When you begin building the back end of your warehouse, tread lightly. Sometimes a cigar is not a cigar, and just because someone says they have a data warehouse does not mean they do. Let's start with a basic example related to one warehouse for which we developed a front end. It contained hundreds of small DBF files. Some data was collected manually and stored in Excel spreadsheets and then converted to DBF files. Furthermore, the design of the individual DBF files set the evolutionary process up for failure.

Let's start with the manual data collection process. This process was a mess. Human error ran rampant. Every time somebody decided to change around their spreadsheet, add a column here, a title there, the programs that converted this data to DBF files crashed. This process also took upwards of two days every month.

Next, let's talk about the file layout. The files were set up horizontally instead of vertically. This may be easier understood with an example. For instance, Company A sells diapers and wants to maintain three years of data in their warehouse. The following is an example of one file:

Filename: **A051996**

Variable List:
RegSmall
PreSmall
RegMed
PreMed
RegLarge
PreLarge
DollRS
DollPS
DollRM
DollPM
DollRL
DollPL
Day

This is a fair representation of the way the company stands today. They carry two styles of diapers, regular and premium, and three sizes, small, medium, and large. They maintain both the number of diapers sold per day and the corresponding dollar amount. Furthermore, the name of the file indicates the date, for instance this file represents "May 1996."

If the company maintains three years of data they will keep thirty-six files (12 months per year X 3 years). The files will range from 27 records to 31 depending on the month. Also, all of the data is numeric, it is either a number representing the count or a number representing the amount.

The first question that should pop into your head, is "Hey, what if they add a new style?" What happens is an unhappy IT person has to go back and add the appropriate variables to all thirty-six files. In our example company A reports on three sizes, for both count and amount so our unhappy IT person has to add six more variables and change all of the programs that need to take these into account.

Another repercussion of adding variables like this, is some data storage devices have a limit on the record length. If there is a limit on the record length, there are only so many styles and sizes the company can maintain by storing this information horizontally.

Let's change things around a little bit. For instance, since we have several sizes for diapers, let's create a variable called "Size" and store the actual size in the field, i.e., 'Small'. Let's also create a variable called "Style" and store the name of the style, i.e., 'Premium'.

We also need to create two more variables called "Amount" for the dollar amount and "Count" for the number sold.

Now our file looks like this:

Filename: **A051996**

Variable List:
Day
Size
Style
Amount
Count

Instead of having a short, fat file, we will have a tall, skinny file. We will never have to worry about exceeding the record length with this technique (that is, unless you store other extraneous information, i.e., at what absorption level leakage will occur.)

Another benefit to this technique is that if we start collecting data on new styles or sizes, none of the files need changing, so to speak, and our IT person is happy again.

For instance, say Company A decides to expand their product line to also include extra-small and extra-large sizes. Nothing in the data warehouse will change, except maybe some transformation and cleansing programs, to handle the new

sizes. They will simply fall under 'Ex-Small' and 'Ex-Large' in the Size Variable.

Another difference is that data can be stored the way it should appear to users on reports or during interactive querying. This eliminates the need to apply formats at run-time. In our example, the performance will probably not be affected because we our not dealing with a large amount of data. However, when data gets to be in the hundreds of gigabytes or even terabytes, every CPU second counts.

What About Summarization?

In this DBF example, there is another problem that can cause terrible response time when doing interactive querying. If a user wants to look at all three years of data, summarized over each year, essentially every table will be joined. This is a common request that a user would make. Joins can be very expensive in terms of time and you cannot make as good a use of indexing.

Adding More Data

The last issue I am going to address in this example is maintaining additional data. Currently, Company A only looks at three years of data. After they have access to the new interactive querying tool, they decide that it would be great to look at five years of data. By maintaining separate DBF files for each month, for every year Company A wants to add, there will be a minimum of twelve new files for every data source. This can cause meta-data and performance nightmares. There is nothing dynamic about this process, it is highly manual. Changes have to be carried over to programs as well as modifying any meta-data structures in place.

A better approach to this would be to keep a "Date" variable instead of the "Day" variable we currently maintain and store all of the data in one, long, skinny file. You might think, "But the file will be so large, we will never be able to manage it!" Now we are talking about data warehousing. When you hear terms like "hypercube," "multidimensional," and "star-schema," these are the things that address those large files.

History

Some of this may seem a bit fundamental, however, these are easy mistakes to make for inexperienced warehouse designers. One reason for this is current on-line transaction processing systems (OLTP). Many OLTP systems date back ten, fifteen or more years, and many are set up horizontally. Since transaction systems are the basis for the data warehouse, many designers make the mistake of copying the same style.

Beyond the Obvious

Once we move beyond the fundamentals, they are some important decisions that need to be made about how to organize your warehouse for fast access. Let's go back to my hammer and nail example. Not all data fits well into a multidimensional structure. If it can't be summarized, it can't be put into a multidimensional structure. In this case, the designer will need to decide if a star schema or snowflake would be appropriate. When I refer to multidimensional structures, I am not speaking in terms of hypercubes. Hypercubes are also an alternative, however, since they summarize across every possible combination of variables, they get large quickly. Hypercubes are unrealistic when you are speaking in terms of very large data warehouses.

The Marketplace

Before SAS Institute released its multidimensional product, we had a client whose data fit well for pre-summarization. They also wanted to use the SAS System to develop and continue building and evolving their data warehouse. In the interim, we developed our own multidimensional navigator by applying object-oriented design techniques utilizing SAS/AF.

In general, the Marketplace is a program comprised of several object classes that allows navigation through summarized and detail level tables, and through a bidding process, determines which table can solve a problem the most efficiently.

The Marketplace is organized around three types of objects, query processors, queries, and resources.

Queries are requests submitted to the Marketplace encapsulated as objects. All the information defining the query such as group-by variables and where clause variables are represented in the object.

A query processor resolves the query, via SQL, DATA Step, etc. The Marketplace currently supports only an SQL query processor. The query processor knows what to look for in a query and how to transform that to work in an SQL code generator.

A resource is the source of information necessary to resolve the query. Resources can be individual tables or entire data marts. The tables can be either summary tables or detail level data. There is no distinction between the two in the Marketplace during the bidding process. It only requires that a resource exist to solve a query.

Another Example

Via a front end tool, a user creates a request, i.e., "Show me the revenues generated from third quarter 1995". The request, or query, is sent to the Marketplace to determine which resource, i.e., data mart or data set, can provide the result. It does this with the use of query processors, which look at all possible resources, and determine first, if they can process the request and second, which resource can process the request most efficiently. The result is returned in the form of a data set and displayed to the user in a graph or report form.

Who Does What!

The following are the core methods necessary for the marketplace to function: "CAN_PROCESS," "ESTIMATE," and "EXECUTE." There are many other methods contained in the classes that make up the Marketplace, but I am not going to cover them.

Can_Process

"CAN_PROCESS" not only determines whether an individual table can solve a problem, but which data mart it needs to solve that problem. Also, we have encapsulated inside "CAN_PROCESS" the ability to not only pre-summarize data but to pre-apply where clauses for common groups of data. For instance, the company for which we initially began development of this design is a manufacturing company. They look at certain product lines a majority of the time, so by applying a where clause to the summary level data, we excluded hundreds of thousands of records from certain tables to ensure fast access to the most common queries.

Estimate

"ESTIMATE" is the implementation of the bidding process. Currently, we base bidding on the number of records in a table, however, bidding can be based on whatever the developer chooses by altering one method. We have considered adding a clustering technique on which to base bidding as well. Clustering is a means by which you sort and index your data to reduce the number of pages retrieved into memory at once thereby increasing the speed with which a query can be solved.

Execute

"EXECUTE" is the method that resolves the query. "EXECUTE" delegates to different code generators necessary to resolve the queries. By delegating to these objects, we can have different code generators for different problems or different data marts. Delegation is a technique in object-oriented programming that allows one object to "delegate" responsibility for a task to another object by giving control to that object until the task is completed.

Patterns

By building this design, we have the flexibility to add and remove data marts or new summary level information at will. We also have built in performance meta-data to help determine which summarized tables are being used and which ones need building.

We accomplished this with the help of several design patterns taken from the following book: *Design Patterns: Elements of Reusable Object-Oriented Software (Gamma 1995).* Patterns guide the developer in structuring applications for change and provide the vehicle for developing successful object-oriented programs.

First, a short explanation of a pattern. The following is a definition taken from the aforementioned book: *A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.* Think of a pattern as a generalized description of a common programming problem written in terms of object-oriented design. The two major patterns used in the design of the Marketplace were the "Composite" pattern and the "Command" pattern.

A Composite lets individual objects and compositions of objects be treated uniformly. This is how we can allow either entire data marts or individual tables to be resources.

The Command pattern encapsulates a request as an object. The command pattern is what allows us to encapsulate the details about individual queries and send them through the Marketplace to be processed and resolved.

AND FINALLY...

Finally, there a few other topics that are worth mentioning. The environment and indexing strategies that you choose can make the difference in your warehouse's performance.

Indexing

Indexing is very important to data warehousing. An indexed data set can dramatically improve the response time of a query. Simple indexing is useful, but composite indexes can make the difference between an immediate response and time

to get a cup of coffee. For example, on 150,000 records, a composite index reduced the time from a minute and a half to about twelve seconds. Indexing does increase the amount of disk space needed for the data warehouse, however, the pay off in speed is well worth the cost of disk drives.

Environments

The environment you choose to develop and store your data warehouse in is one of the most important decisions you have to make while attempting to complete this impossible task. It can make the difference between success and failure. The Institute has released a new product call the Scalable Performance Data Server® (SPDS) that runs on the Sun Solaris operating system for Unix. This essentially allows for parallel processing to occur in SAS that is virtually invisible to the developer.

Parallel processing is essential on very large data warehouses. No matter how well you organize your data and clean it up, if you try to process several gigabytes of data without multiple processors, you will have time to drink a whole pot of coffee every time you submit a query. If you are operating in a batch environment, this may be acceptable, however, if you are trying to provide interactive access to data, it is entirely unacceptable.

Also, it is essential to have a separate data server entirely devoted to your data warehouse.

WHAT ABOUT THE FRONT?

The user interface is the visual representation of the data in the warehouse. No matter how well you structure your data warehouse, if the user does not have an easy-to-use interface, structured to make changes quickly and cost-effectively, the warehouse will die. It has to have a well-designed foundation to allow modification to existing functionality. Achieving this takes some investment initially, but the payoff can be incredible.

Objects Again!

It is important, from the very beginning, to design every tool with the idea that some day, it will either change drastically or will go away completely. (Remember the most dynamic tool of all is your user!) Therefore, it needs to be easily changeable or removable from the rest of the application. When you drop a widget on a frame, determine how it relates to the overall picture of that frame or even the entire application. Decide what other tools or widgets it will affect. Keep the widget as decoupled from any physical aspect of the system as possible.

One pattern that addresses this problem is called "Observer." It is also known as Model-View-Controller, a term coined by Smalltalk programmers several years ago. The idea of this, is when changes are made to a widget (the view) that affect other parts of the application, the changes are not made directly to these other pieces. They are made through a non-visual object (the model). The model then sends out a message to any part of the system listening for that message that something affecting them has changed. When a listening object hears the message, they perform some predetermined action, for example, a "_REFRESH_."

The benefit of this technique is objects can be added and removed without affecting existing pieces of the application. The benefit in the design of OLAP tools is that parts of tools can be changed or removed without making major changes to the entire application because all interaction occurs through the model.

Also, whenever possible, dynamically create your widgets at run-time. By creating your widgets at run time, you have the utmost flexibility to change and add new features without having to create new frames.

The SAS System allows you to position your widgets and add attachments programmatically. Attachments are an important, and often neglected tool. Attachments allow you to design your interface to run in any screen resolution. This is always important, but especially when you are developing for a distributed, multi-user environment.

Composites

Composite widgets are also important when designing interfaces for change. Composite widgets allow the designer to build a custom widget derived from existing widgets. This is a powerful feature of the SAS System. Developers can build complex widgets that are reusable and easy to dynamically place on a frame. Another reason to use composites is related to how the SAS System uses resource entries.

Resource Entries

The use of resource entries is what allows the frame to store pointers to the classes that contain the definition for each widget. (It should be understood, that a widget is also stored as an object.) Resource entries contain aliases that identify the type of object this class represents. For instance, a graphics widget has an alias of "Layout."

Along with the alias, when a widget is stored in a frame definition, the type is also stored. A command push button,

for example, has an alias of "Gbutton" and a type of "Composit." By only storing a two-level name, the alias and the type, i.e. "Gbutton.Composit," the developer can switch resource entries at run-time by changing the search path. Several patterns in the Design Patterns book use this feature, the "Facade" pattern and the "Factory" pattern are two examples.

Andrew Norton has written a paper called "Object Interfaces" that goes into detail about this technique.

Caching

Another useful technique to use when designing and building OLAP tools is "caching." That is, storing a result and reusing it until it changes. This technique can save a tremendous amount of computation, reducing the total time it takes to return the answer to a query. For instance, when drilling down through an application, storing the results on the way down will eliminate the need to recompute the results on the way back up.

It Really Works!

We installed a first version of a system for our manufacturing company, and after some testing, they discovered that the query building screen we initially built was too slow and cumbersome. They also had additional features they needed added before rolling the application out to their entire user population. We also added additional functionality to the marketplace.

The changes they requested were not trivial, however, we delivered them in only a few weeks time, at a minimal cost. Ripple effects were minimized because of the object-oriented foundation that had been laid for this project. Definition: *A ripple effect is a bug that occurs in something that was working as a result of fixing a bug in something that wasn't working.* This does not make for a happy programmer.

CONCLUSION

Business has been the driving force behind the data warehousing initiatives, not academia. Why? They need information faster and more reliable than ever before. However, data warehousing is an expensive undertaking for businesses, and if you don't get it right the first time, you may not get a second chance. Objects let you get it close the first time, but give you the flexibility to continue to improve and eliminate mistakes through iteration.

Building evolutionary systems has always been a goal, however the technology historically has not been available. Today, we have the technology; now we need the training and business understanding to build successful, progressive systems. Data warehousing requires evolution, evolution requires flexibility, and flexibility is provided by objects.

ACKNOWLEDGMENTS

Thank you to Andy Norton, without whose mentoring, I would not be writing this paper and whose ingenuity crafted the marketplace design.

Amy Turske McNee
Trilogy Consulting, Incorporated
5278 Lovers Lane
Kalamazoo, Michigan 49004
(616) 344-4100
akturske@trilogy-cnslt.com

REFERENCES

Gamma, Erich, Richad Helm, Ralph Johnson, John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company.

Norton, Andy (1997), "Object Interfaces," in *Proceedings of the 22nd Annual SAS Users Group International Conferenc*. Cary, NC: SAS Institute, Inc.