RECORD OF SOCIETY OF ACTUARIES
1995 VOL. 21 NO. 1

TECHNOLOGY NEWS FLASH

Moderator: ROGER W. SMITH
Panelist: MICHAEL F. DAVLIN
Recorder: ROGER W. SMITH

What new technologies have actuaries been implementing lately? What technologies have special interest to actuaries? Speakers will describe how new technology has been implemented and how technology could be implemented in actuarial operations.

MR. ROGER W. SMITH: This is a panel discussion on hot technologies. These sessions for the Computer Science Section are very much like tax law and valuation law update sessions. There's always something to talk about. Joining me on the panel is Mike Davlin who is with Actuarial Resources Corporation and I'm with PolySystems. Mike has been very active in actuarial roles and in technology for quite a number of years. In fact, I speculate he might be the answer to a trivia question. Many of us were first introduced to Shane Chalke through a paper on the universal life valuation model. The coauthor of that paper was Mike Davlin, so he has been around quite a while.

The topics I will be discussing are large data sets, PCs, and networks. I decided to talk about that after a couple of very large disk systems went south on me. I got smarter after that point. Now, let's hear what Mike has to say.

MR. MICHAEL F. DAVLIN: Roger and I go way back as he mentioned. We first met in the late 1970s when I was a young reinsurance student with the Transamerica Companies. For 12 years I was involved in quite a few traditional actuarial roles, but I always had a long-standing interest in technology. I had a torrid, 12-year love affair with A Programming Language (APL), as most actuaries have had at one time or another. But when the PCs came out, mainly for reasons of performance, I found my interests broadening. Gradually, I wanted to become more deeply involved in software development, so I left TransAmerica in the mid-1980s and joined Tillinghast. I wasn't able to get as involved as I had hoped in software development there, so I founded Actuarial Resources Corporation (ARC), which now has several offices across the country.

At ARC, I started working with dBase and dBase compilers, and then moved on to Modula 2. From Modula 2, I moved on to C and 80x86 and 80x87 Assembler after I read an article by Bill Gates on what would be required to write in Microsoft Windows. This was in 1986. From that point on I became deeply involved in DOS extender technologies. My interest in databases and DOS extenders led to a three-year partnership with a bank consulting firm where I headed a team that developed large software applications for banks and savings and loans. These applications were a hybrid of data warehouse querying systems with financial simulation capabilities layered on top.

I returned recently to APL development, as I spent a couple of years heading up software development for Chalke Inc. in Virginia. I probably know more than anybody would ever care to know about hybrid software development using APL, C, and Assembler. If any of you use PTS, you have probably used the Intex CMO modeling add in, or maybe you've noticed there are some assembler speed-up routines in the PTS asset model, or the entirely new user interface in version 7.3. My old company, Davlin Consulting, developed the technology required for all of that to work properly.

I've done a lot of work on securities pricing engines over the years, and have developed a variety of actuarial applications as well as some nonactuarial applications like network routers—a product which allows one network to dial into another network so a user on the first network can access a file on a second network across the country as though it's a local network. I've also developed medical and seismic imaging systems, and, as I've already mentioned, banking applications.

I want to talk about the banking application because it ties closely into my first topic. We wrote a lot of Common Business Oriented Language (COBOL) routines on the mainframe that would extract data from a lot of disparate data sources that banks used. We coalesced all of that data into one data warehouse. We targeted 386 PCs, and at that time, we thought we were dealing with inordinately large data sets. We often handled up to a gigabyte of data. Using 32-bit DOS extender technologies and C with some assembler optimizations, we were able to write software that banks could use to query and report on their contracts. Simultaneously, they could write up to 300 reports, each of which had its own unique filter applied to the database. They could process the records at about 125,000 records a minute, which was phenomenal at the time. That software can process in excess of 500,000 records a minute on today's hardware. We had demonstrations that I'm still proud of where the data processing (DP) people would walk around the table on which our laptop was sitting and look for hidden wires because they couldn't believe a laptop could process data so quickly.

Since then, the bank consulting firm has rewritten that entire product line to bring it up to the modern Graphical User Interface (GUI) and client/server age. They went to Windows using C++, adopted a client/server, relational database technology that enabled them to raise the price of the software dramatically, because that's what everybody wanted. But the performance fell by almost a hundredfold, which gives you an idea of the performance penalties you can incur for the ease of use and security of relational technology.

At that time, I felt that there had to be some third way between a totally proprietary file format which nobody could access as an end user with standard report writers and other tools like spreadsheets, and fully relational systems that open the system up for the user. But, in my view, things are just far too slow for routine querying and analysis.

So the first thing I will talk about is a database technology that is actually old, but is having a comeback: multidimensional databases. It's also often referred to as On Line Analytical Processing, or OLAP. I'll talk about the distinction between physical and logical OLAP implementations, and briefly mention something called the OLAP Council, which is trying to standardize this technology in the same way that relational databases and Structured Query Language (SQL) were standardized.

My second topic will be component-based software development. My current firm is very excited about the direction Microsoft is going with something called the Component Object Model (COM), which is now the new low-level foundation for Object Linking and Embedding (OLE) within Microsoft Windows. OLE is a mechanism that has grown quite a bit since its first incarnation as something that allowed you to embed a chart in a Word document or Excel spreadsheet. I'll start with the multidimensional database technology: what is it, what are the alternatives, and why in the world would you choose to use one?

Before I start in earnest, I want to mention a new buzz word in the OLAP marketplace: it's BIS, which is short for Business Information Systems. It's an attempt to marry the best of both the Executive Information System (EIS) and Decision Support Software (DSS) concepts. You have probably heard of EIS applications, which generally entail a GUI presentation layer, point and click data navigation, and a hierarchical presentation of data with a lot of consolidation of information. This is combined with the ability to drill down into lower levels of detail. Another feature is building in alarms and triggers so that certain relationships in the data are automatically brought to your attention if it's deemed to be important according to your business rules.

The second category of software that BIS draws upon is DSS. Originally DSS was mainframe based. It was largely used by econometricians, statisticians, and it had a heavy emphasis on time series analysis. A unique aspect of early DSS products was that normally their file formats were explicitly designed to support a multidimensional data cube; I feel this is really home base for actuaries, especially those working with APL environments. I think most actuaries, and other financial analysts, tend to view their business and their data as a multidimensional array. Later, I'll discuss in more detail multidimensional databases that give you that type of a view.

So why use multidimensional databases? Well, I already mentioned the first reason, which is that with multidimensional databases, the primary logical idea that it presents to you is a data cube, and that tends to conform to the way financial analysts look at their organizations and related data. If you will be doing financial analysis or statistical analysis, as opposed to On Line Transaction Processing (OLTP) where you might want to update a record to reflect a premium payment when it comes in, a multidimensional database will far exceed the performance of a relational database. In order to use it, there's no or very little programming. In terms of flexibility, the OLAP product's flexibility is a lot greater than it is in relational products I've used. The multidimensional database is complementary to existing relational database systems; it doesn't have to displace them. Properly designed, it can relieve a lot of network congestion in comparison to data warehouses that are designed with relational database technology.

One thing I really appreciate is that they have direct support for time hierarchies. By this I mean that an OLAP product treats time as a special dimension. It knows that days roll into weeks, weeks roll into months, months are going to roll into quarters which might be calendar or fiscal quarters, and it understands the difference between calendar years, fiscal years, year-to-date-types of balances, quarter-to-date; that's all there for you. Combining all its features, it really facilitates what the OLAP advocates call data surfing—being able to bring up all the data pertaining to your organization onto your user screen, and cruise through it looking for anomalies or explanations for whatever problems or patterns you're trying to explain.

The history of database architectures can be placed on a time line. The first files used were flat or indexed sequential access (ISAM) files. These weren't particularly adept at representing the hierarchical relationships—parent/child types of data relationships—that you routinely encounter in accounting systems, projection systems, and other financial applications. Over time, people found that a record frequently could or should be a member of multiple parent/child hierarchies at the same time. For example, a plan of insurance might be a member of a hierarchy that defines reporting rollups in a corporate projection model, while at the same time it might be a member of a hierarchy that represents reinsurance contractual relationships within the same model.

In the late 1960s, there was a committee formed called the CODASYL Committee whose charge was to develop a standardized approach to managing these simultaneous hierarchical relationships. It came up with a database model that we used in our banking software called a network model database. It is very good at representing data relationships that aren't simply parent/child: for example, relationships that are more complicated, like a three dimensional web or graph of data. One of the problems with the network database model is that its file format is very intricate and contains many physical offsets of records in other records (sometimes these are logical offsets, but problems remain) for use as addresses for fast navigation through the database hierarchies. If you are unfortunate enough to get any file corruption, you're dead in the water, as many unhappy people found out. Sometimes a database repair program could minimize the damage, but too often a complete reconstruction was necessary. Another drawback of the network model database is that users and programmers have to understand the logical and physical implementation of the database in order to navigate and access its data and hierarchies. Nonetheless, the performance advantages of the network model often outweighed these drawbacks.

Partially in reaction to this, an IBM computer scientist named Dr. E.F. Codd invented the relational database model. A relational database is comprised of many two-dimensional tables, rows and columns, which makes these tables very similar to the familiar spreadsheet. Relational databases are often referred to as Relational DataBase Management Systems (RDBMS) products. Viewed abstractly, the relational model has many nice mathematical properties. In fact, there's a fully elaborated mathematical theory behind relational databases based upon set theory. I think that is one of the reasons computer scientists like the relational database model so much. In the same way APL appeals to actuaries, the relational model is mathematically very appealing to computer scientists. Dr. Codd came up with 12 rules to which a "truly" relational database must comply. Even today, I don't think there's a single relational database on the market that meets all of Codd's rules; clearly, I think he was a bit of a perfectionist. I mention this because Dr. Codd will show up again in the OLAP arena.

More recently, I think you might have heard about object-oriented databases, or Object Oriented Database Management Systems (ODBMS) products. These tools are great if you're developing in a language like SmallTalk or C++, which are inherently object oriented. But, in terms of BIS applications, it still has the problem that the conceptual view of the data in the database really doesn't match the end user's view. Most recently, according to many analysts, the up and comer for BIS applications seems to be the multidimensional database model. Currently in terms of implementations, the dominant databases in the relational camp are Oracle, Sybase, and Microsoft's SQL Server. I'm sure some of you have at least one of those products installed somewhere in your offices. In terms of object-oriented databases, probably the three leading by number of installations are ObjectStore, Versant, and Poet. These are probably all foreign to you unless you are deeply into C++ or SmallTalk development. I will not dwell on these. I just threw them in so you would know they're there. Some of these other products may be more familiar to you. I know a lot of actuaries develop in dBase, Fox Pro, Paradox, Access, and Alpha Four. These are not truly RDBMSs, and many experts call them near relational systems. They're a lot like relational databases in that everything is designed in terms of two dimensional flat tables. They have indexes and all the usual paraphernalia, but for a variety of reasons, they don't come close to meeting Dr. Codd's rules as full-fledged relational databases.

There are some real advantages to relational databases, and I think that's why they're used so frequently today. One big one is they're standardized through the American National Standards Institute (ANSI), and they are very easy to understand. By now nearly everyone understands a spreadsheet, and when you design or look at any particular table in a relational schema, everything resides in a flat, two-dimensional table just like the tried and true spreadsheet. It's attractive to DP professionals because they know that if they have a properly designed schema, it's a logically complete database, and you can design a query that can answer virtually any question you want to throw at the database. However slow the answer may come back, it will come back. It has programmer and end-user query language that is very standardized through the efforts of ANSI and others called Structured Query Language (SQL), which is usually pronounced as "sequel."

Recently Microsoft defined a new SQL-based protocol for accessing data in relational databases called ODBC, which stands for Open Database Connectivity. I don't know if any of you have heard of this before, but it's a de facto standard that was designed and aggressively promoted by Microsoft. It tries to abstract away all of the differences in SQL implementations between various relational databases. With ODBC, which looks a lot like SQL, you can write an application that makes ODBC calls for all of its data. Later, if you desire, you can replace your back-end relational database without changing your application source code. This enables software developers to develop one version of their front ends that can work with many different back ends. If you make all your calls to ODBC, you might have it running on an Oracle database, and if you decided you didn't like the performance, you could switch over to a Teradata database running on specialized multiprocessor hardware, and everything would work with minimal changes to your application. The last big advantage to relational technology is that it's very mature, there's a lot of vendors, many supporting products, and there are a lot of developers for hire who know how to design and use them.

There are some disadvantages to relational technology for BIS applications. I already mentioned one several times: performance can be a real problem. The way people generally try to work around that is through something called database denormalization. Usually when you design a relational database, you try to design the tables in such a way that you minimize data duplication. Unfortunately, in an application of moderate complexity, that usually results in hundreds of different tables. When you want to run a query that might bring pieces from different tables together, you have to do something called a join. Each of these joins can create a temporary table, and sometimes it can take forever for all this to happen. Another thing that might be done, especially if you're trying to do hierarchical rollups—maybe through your chart of accounts or your reporting hierarchy—is to precalculate some of those values. This can help quite a bit. Relational database vendors have created specific proprietary programming Application Programmers Interfaces (APIs) for C programmers, or developed proprietary extensions to SQL which are nonstandard. These tricks can sometimes improve performance significantly. Probably the most common solution is to spend a lot of money on specialized, expensive hardware. There are very fast multiprocessor platforms designed exclusively to serve as back-end relational database servers that can range in price from $60,000 to millions of dollars. These can really improve your performance, but I think there's a better solution.

As I mentioned, many data relationships that you encounter when you create financial modeling software are graphical as opposed to hierarchical: data can reside in many hierarchies at one time. It's difficult to map graphical data into a two dimensional format, so you wind up with very convoluted relational database schemas. Once you've done this, and you want to query and get data back as an end user, as a financial analyst, you have to have intimate knowledge of the database schema; it's very easy to make mistakes. Very often you can do an incorrect join of different tables and get data back, but you don't realize you have made a mistake, and you have bad data. There's also no native built in support for time and its role within reporting hierarchies. I view that as a real disadvantage of relational technology: you have to build all that support for time explicitly yourself.

Object-oriented database management systems are about ten years behind RDBMS products in terms of the product life cycle. They seem to be showing a similar S-shaped growth path, and a lot of people think they will overtake relational systems perhaps five years from now. There's another camp, led by C.J. Date, that feels we can simply change relational databases by just taking a column and letting that be a generalized object instead of being restricted to a fixed set of values, like integers and text. Some people think that the two approaches will be married in this way, by generalizing the notion of the data a relational column can contain. Another camp envisions hybrids developing from the other direction.

There is an ODBMS industry working committee called the Object Data Management Group, or ODMG. It has come up with something called Object Query Language (OQL), which is an ODBMS counterpart to ANSI SQL. OQL is becoming standardized and supported in most ODBMSs through the efforts of the ODMG. If there is any data that looks like a table in an object-oriented database, OQL gives you a simple query language that is built on top of ANSI SQL with some extensions. You'll be able with an object-oriented database to get into an end user product that supports third-party report writers, things like Excel or Microsoft Access, or into a separate product like Crystal Reports, and write your own reports against your object-oriented database as simply as you can now write SQL reports against your relational databases, but I am getting ahead of myself.

One of the main advantages of an object-oriented database is the following. If you have a graphical data relationship, you will usually represent it as a kind of a graph in memory. When you want to write that complex structure out to a disk in an ODBMS, it looks the same way on disk as it does in memory. You don't have to break it down into a bunch of different, but related, tables. This is often referred to as the object to relational impedance problem. An analogy some people use is if your car was your object and your garage was your relational database, to get the car into the garage you'd have to take the wheels off and find the right cabinet to put the wheels in. You'd put the wheels in, then you'd take the doors off, find the right cabinet for the doors, put those in, etc., until you have the car entirely disassembled into its constituent parts. When you want to use the car, you'd have to reassemble it by doing a bunch of queries to get all of the constituent parts out of the cabinets. In an object-oriented database, you just put the car in the garage—that's it. It solves tedious programming problems from a developer's standpoint. Because of that, it can be up to 100 times faster when accessing data that has this kind of a graphical relationship; which again I think a lot of data have—especially actuarial modeling systems.

Another advantage of object-oriented databases is they tend to have built-in support for something called object versioning. The idea here is that you have an object, maybe it's a corporate model, and you save it in the database. Later, you want to keep it under the same name and bring it out to make a change to it and save it back. With versioning, your current version is in there, but the earlier version is there too, so you can roll back to an earlier version of your model. You could go back the last year-end by pulling up the correct version of your corporate model, make a minor tweak to it, and save it. Again, you still have both versions so you can easily rerun both versions to identify the effect of the change. Another nice aspect of object databases is that most of them support local workspaces. It's something they've thought through fairly well. The idea is if you have a large object-oriented database, you might want to work with a subset, maybe you want to put it on your laptop and take it home for the weekend. This feature allows you to do this with the facility to merge your work back into the original database.

Object-oriented databases have their disadvantages. They're not yet standardized to the degree to which relational databases are, but that seems to be coming. There's far fewer credible vendors than for relational databases. There's far fewer supporting software utilities. Relational databases have it hands down over object-oriented databases when it comes to supporting utilities. It's early technology; everybody's trying to make a lot of money, so licensing is very expensive. ODBMS APIs tend to be very language specific. Usually, they're designed for C++ and SmallTalk, so if you develop in COBOL or APL, the object-oriented databases at this stage probably aren't a viable choice for you. In terms of business information systems, there are drawbacks to both the relational and the object-oriented database. In both cases, the database schema, the way the data looks in the database, and the programmer's API to get it out don't conform to the analyst's mental model of his data. Analysts have to be able to program in order to manipulate the data, and because of that, the analysts have to be very familiar with the structure of the data. While both models tend to work fairly well for static views, they are less appropriate if your data will change frequently, especially if you will change the way you want to roll things up. So after denigrating those two models, I'll try to explain to you what multidimensional databases are about.

I think multidimensional databases are ideal for many kinds of actuarial analysis: experience and expense studies are obvious examples. I don't know if you saw the most recent *Financial Reporter*. There was Part I of a three-part article on different costing methods [Blake, Jane, "New Cost Views: Part I. What Every Actuary Should Know?," *The Financial Reporter*, March 1995, no. 25, pp. 1–6] and there was one called activity-based costing. I know there are several companies that use these specifically for that purpose. They'd be great for actuarial assumption browsers. Generally, actuarial assumptions are inherited according to some sort of a hierarchy. Take, for example, agent commissions. You have a base set that applies to just about all of your life products, but it might vary a bit at the upper ages. You might also have a special set for a few plans. You can naturally represent your company's commission structure as an inheritance hierarchy in a multidimensional database. The same thing applies to your mortality assumptions and plan features. OLAP products are great for browsing financial and valuation statements. They've been used in budget and planning applications, and I think it would be really useful for browsing multiscenario simulations. That's an area where I think a lot of modeling systems really fall down. They do many simulations, leave you a lot of data on your disk, but there's nearly nothing you can do with it afterwards.

OLAP is superb for analyzing sales data. A lot of retailing firms use these types of databases. Mervyn's, in particular, has really gone hog wild on OLAP technology as they believe it gives them a real competitive advantage. Just to give you an idea of the capacity these systems can support, Mervyn's has right now 700 gigabytes of data on line, so this truly isn't a toy technology. You can put anything from a few megabytes all the way up to a terabyte and more in these systems. Another feature that's very nice in some OLAP/BIS products, and this is borrowed from the EIS camp, is that you can program in alerts and alarms. This would be very useful in valuation systems. You can carry historical valuation data even if you want to do so at a seriatim level (if you want to buy the disk base to support it) and build in automatic triggers for alerts and alarms for patterns that look funny. This way you're not surprised after working all weekend and slaving to get your numbers together when you take it to your boss and he or she immediately points to the one number that looks odd. You can get warnings on things like that. Another thing that not all, but some of them support is a built-in language for doing time series and econometric types of analyses. This ties back into your experience studies. If you feed in much of your transactional data, you can perform mortality and other decrement studies, and analyze premium payments and other policyholder behavior with these systems. A product called Acumate from Kenan Technologies is one such OLAP product that has extensive support for financial and time-series analyses.

There are two main categories of activity you'll read about in the literature on databases. The first is something called OLTP, which stands for On Line Transaction Processing; that's a well-accepted relational database term. The other is OLAP, which I've already mentioned stands for On Line Analytical Processing. Relational databases are very well suited to OLTP processing; in fact, they are optimized so they perform well on standard OLTP benchmarks. They do not perform as well for OLAP work. Dr. Codd, the father of RDBMS technology, recognized this fact and dubbed another access method OLAP. He wrote a paper called, "Providing OLAP to User Analysts and Information Technology Mandate," wherein he coincidentally created 12 rules an OLAP server should comply with—all databases naturally appear to have 12 rules. I'll come back to these in a little while.

Earlier, I mentioned the OLAP Council. This group was formed after the OLAP market went in a dozen different directions over the last four to five years. It finally dawned on most of the OLAP vendors that, if they were to effectively compete against relational technology, they would have to get their collective act together and provide some standardization of features and APIs. So the OLAP Council was formed, and I have a few quotes from them. The Council feels OLAP's most important attribute is that "OLAP is for users, and users think multidimensionally." However you, as an OLAP server, store the underlying data; the view of the data you present to the user must conform to the multidimensional conceptual view of the user. Further, they define OLAP as "software technology for transforming enterprise data into multidimensional information to support end user, interactive, exploration, and analysis for better decision making."

Now, in general, OLAP products have the following characteristics. Whether it's implemented this way or not when you set it up, it looks to you, the end user, as though you have a multidimensional data cube. You can pick however many dimensions you want, and which of your data items are the dimensional items, and which are other fields, sometimes called metric fields, that hold what they call the facts or measures. Typically, things like premiums would be a measure field; dimensions would be data items like sex, smoker or nonsmoker, risk classification, line of business, plan of insurance, etc. One unique thing about these OLAP products is that you can take your dimension fields and construct hierarchies along the dimensional fields. You could use face amount as a dimensional field, and then break it into bands as the next hierarchical level. The same thing with account values, you can tier your account values. You could take your state codes and roll those up into regional and different distribution hierarchies.

All the OLAP products I have reviewed give you the ability to easily pivot the cube. Excel's pivot tables are really great, and I wouldn't be surprised if the inspiration for those came from the OLAP concept and products. The only drawback to Excel pivot tables is that they are limited to a volume of data that can fit into available memory. While Windows utilizes a virtual memory scheme, triggering it would cause a large volume of paging to and from your hard disk so your response time would be very slow on large data sets. While OLAP products provide similar functions to pivot tables, they have the advantage of being designed from the ground up to handle data sets of up to a terabyte in size. If you set up 11 or 15 dimensions for your cube, you can rotate the cube in whatever way you want. If you decided you wanted to look at business by plan and then state underneath that, you can easily rotate to that view just by dragging your mouse so that plan is first and the state is under that.

OLAP systems have the ability to define consolidation paths with drill downs. By consolidation paths, I mean primarily those attribute hierarchies along your selected dimensions. Most OLAP products tend to precalculate and preaggregate a lot of data so you can start at the top and drill down. If something looks funny, you can rotate it, filter it, or basically do whatever you want, including creating multidimensional cross tabulations. Some of you may have seen a product called Forest & Trees, for example. It was an early EIS product. It allowed you to do cross tabulations, but they were just two dimensional. OLAP products allow you to do five, six, or seven dimensional cross tabulations. One of the nicest features is that you can nest the rows in the columns so you can get a two dimensional display of something that's really a five or a seven dimensional cube. In other words, you might have year, quarters under years on the column, and budget and actual, so that gives you three dimensions. You also might have region totals and then break down your rows by state. And again, you can easily drag one dimension that happens to be on your columns over the rows if you want to flip things around. They all allow you to apply filters to select subsets of data. In other words, even if the logical structure of your data cube is defined and fixed, you can put a filter on the view such as "give me all the business between a certain date with a face amount in a certain range" and whatever else you want to filter on. It has a native support for time which I really like, and again it has point and click data navigation capabilities.

There are two diametrically opposed approaches to implementing these kinds of products. One camp says that if you're going to have the performance, you have to have a multidimensional physical implementation. You can't use a relational database as the data store for the OLAP server because it will be just too darn slow. And so, some companies have gone in that direction, but a lot of DP shops really don't want to have the expense or the consistency problem of having duplicate data.

There's another camp that says you can provide what's called a virtual or an abstract OLAP to the end user. For example, if they want to drill down a level, you can construct a SQL statement, issue that SQL statement out to a relational database server, get the results back, and then map it back into whatever multidimensional or cross tabulational view the user was expecting when you place it on the screen. That works too, and many companies have gone that route. It can be a good deal slower than the proprietary data formats, so other more recent products have actually found a middle ground. These products rely on the premise that users rarely drill all the way down to the lowest levels of their data cubes. If that is true, then the following strategy makes sense. Picture your N dimensional data cube; there might be 11 dimensions. Down to maybe six or seven dimensions, they'll store summarized and aggregated data in their own proprietary format. But, when you want to drill down below that level, they'll switch to a mode where they issue SQL statements out to the original RDBMS data source to fulfill that query. Given that RDBMS databases often reside on faster machines with wider input/output (I/O) channels than the typical OLAP server, performance may not suffer at all, and could even improve.

As to which approach is best, like anything in systems work, it's all a matter of trade-offs. The proprietary multidimensional database format is much faster, holding hardware constant. But the cost is that you duplicate data, so you need more disk space. You have to import the data into the multidimensional database—that happens at some frequency. You can control it, but that's a potentially time-consuming process. On the other hand, if you go the route of just providing a virtual OLAP layer on top of a relational database, you run the risk of clogging your network with a lot of SQL traffic. The way that works is you have your SQL server, it returns a lot of raw records back to this virtual OLAP product and on the client machine, it's doing a lot of assembling to make it look like it's an OLAP server. Whereas, if you actually have a multidimensional database installed on the server, much of that activity happens on that machine and just the multidimensional or cross tabulation view comes back to the client machine over the wire.

Now let's go back to Dr. Codd's 12 rules for OLAP. Codd says that anything that wants to call itself an OLAP server, has to have a multidimensional conceptual view. The second rule is transparency. Codd says it has to have an open front end. One of the nice things you'll note about these products, if you investigate them, is that they're not monolithic products with their own front end, that you have to use. A lot of them are designed such that they're either in the form of Dynamic Link Libraries (DLL) or Visual Basic (VBX) controls so you can design your own front end and make calls into these servers. Many are working on the next generation of components: OLE Custom (OCX) controls. You can stay in an Excel spreadsheet, for example (many of them have Excel add ins), and you can connect to the OLAP server from these familiar products. You'll get a spreadsheet that's the top layer of your data, if that's the way you set it up, and then you can start drilling and rotating to your heart's content. Once you've hit a level or view you want to analyze, you can do whatever you normally do in Excel; it's really quite slick.

Accessibility is another criterion, and the idea is that if you're going to be an OLAP server, you have to be able to take data from a relational database, flat files, and hierarchical databases, etc. It has to be very open-ended on the back end.

Consistent reporting performance is another rule. Most of the individuals involved in the OLAP marketplace feel that if you initiate a query, you shouldn't have to wait more than a few seconds or a minute at most to get the results back; so that's the type of performance they're looking for and you should expect. It should be a client/server architecture—again, for a variety of reasons; it's mainly for cutting down network traffic.

The idea of generic dimensionality is that there should not be any privileged dimensions. You ought to be able to use any of your data fields as a dimension if that's what you want to do. There shouldn't be any built-in constraints with regards to choice of dimensions. If you construct a logical data cube that has 15 or 16 dimensions, you may find that it has many potential cells. If your database system preallocates space for all of the potential cells, you will find yourself in trouble. This is why Codd stipulates that one of the key features to be supported is sparse matrix handling. You only will use disk space for parts of the data cube that are actually occupied by the data that's in your organization. OLAP products should also have multi-user support.

Another rule is intuitive data manipulation. I think what Codd meant is that, by and large, the interface should be graphical, mouse driven, and very easy to use. Flexible reporting—there shouldn't be any artificial limits on the number of dimensions and aggregation levels you wish to define and use.

If you're at all interested in this technology, you should contact the OLAP Council that was formed last year. It's an educational entity to promote this idea. What it's trying to do is standardize the semantics. Originally everybody had different words for dimensions and measure fields and the operations of rotating the cube or applying a filter. The other thing it's trying to do is standardize an application programming interface (API) for OLAP clients. And again, the reason that's important is the same reason ODBC is important for relational databases. As developers, the last thing you want to do is have to write a special version of your application for every relational database that you care to support or your users want to use.

The same thing is true with OLAP servers, and so what the Council is trying to do is to find a standard API so you can write one client application, market it, and your clients can purchase whichever multidimensional database servers they want. These APIs will build on something called OLE 2 objects. I will go over that later, but for now you can consider an OLE object to be a self-describing object that exposes server data and functions, a slice of that data cube, to client software. In other words, it's an application that you or I write. Benefits of this standardization, if it happens, will encourage many third-party developers to write even better querying front ends for these products. To be honest with you, much of the emphasis has been on the server side, not so much on the client side of the picture, the side you would use as a financial analyst. Standardization should reduce the learning curve. The common semantics, common API should make it easier for both developers and end users to learn how to use these products.

Just to give you some assurance that this isn't technology from Mars, there's a research company called the Meta Group. In 1994, it did a survey on data warehousing. Data warehousing is the idea of taking disparate data sources throughout your organization and coalescing them into one central store. The goal is common, consistent data. It might be housed in a relational database or it might be stored in something else. In 1994, it surveyed large organizations, and 95% had data warehousing plans in 1994 and 1995; more than 50% at the time of the survey already had data warehouses in production. Their investment plans over the next 12–18 months were that 29% intended to spend between $1 and $3 million on data warehousing, and more than 50% planned to spend less than $500,000. In terms of corporate areas surveyed, it was 26% customer information, 24% marketing, 22% finance, and 16% sales. In terms of the scale of their endeavors, the initial data warehouse size tended to be less than 10 gigabytes. I don't know whether 10 gigabytes seems like a lot of data. I guess I'm becoming immune to it; it seems small to me. The survey indicated that projected size, 12–18 months out, will be 20–100 gigabytes of data. The initial number of users would be less than 50; projected number of users in 12–18 months would be 50–500. In terms of the back end, and some of the relational databases that people were tending to use, many of them used IBM's DB2, but Oracle was probably the most common. This is what I found interesting. Sixty-five percent of the surveyed respondents said that some sort of a multidimensional or OLAP layer on top of the data warehouse was very important or important.

FROM THE FLOOR: To access all these multidimensional databases, will you need code? Is that where you will use your C++?

MR. DAVLIN: That's right, you could use C++. Many of them come with a standard query package, but you could use C++. Other ones, as I mentioned, use either DLLs or VBXs. So, you could develop in any language which allows you to call into DLLs, VBXs, or soon, OCXs. Also, most vendors offer add-ins for Excel and Lotus. You can be very eclectic with your tools and still use this technology effectively.

FROM THE FLOOR: Are you talking about keeping insurance companies perhaps on these systems?

MR. DAVLIN: Not in the administrative sense; it would be more of a data model. I would put all of your in-force records and many of your transactions in there, along with competitor information. This way you can do correlation and policyholder behavior studies and things like that.

FROM THE FLOOR: Someone would be writing code, wouldn't they?

MR. DAVLIN: At a minimum, someone would have to design the schema of your cube, and get the data in there. But, beyond that point, there's not a lot of code to write, because most vendors provide generic, off-the-shelf data analysis and navigation tools that will more than suffice for most users. Generally, what you get is something that looks like a spreadsheet: you get your list of dimensions and at that point, you're drilling down into dimensions or setting up cross tabs or applying filters, for which very little or no programming is necessary. If you want a highly tailored, customized application, then you would turn to the component libraries and break out your favorite compiler or interpreter.

I'll try to cover my next topic, which is Microsoft's efforts regarding the COM and OLE. If you use Excel or Microsoft Word, you have most likely heard of OLE, which allows you to link or embed a document or object from one application into a document created in a second application. OLE originally was built on top of something called Dynamic Data Exchange (DDE) which has had a long history of being slow and somewhat unreliable. As of the time OLE version 2.0 was released, OLE was built on COM.

COM is a software specification for creating standardized, reusable components. The COM architecture is supposed to allow binary components supplied by different vendors to interoperate in a reliable and controlled manner. Historically, what we've had for applications, whether we're talking about actuarial software or general PC market software, is a lot of monolithic applications. They're all built in one giant piece; any services they need other than from the operating system are embedded inside the application itself. Since Windows with tools like DLLs and VBXs, we started to see applications that are built from components: not all of the functionality is in the application itself. What's going to happen next—and you can see it a little bit in Microsoft Office if you use Microsoft Office—is that applications themselves will become components. Other applications can drive other applications, and this will change things.

As I mentioned, a monolithic application might have had 100,000 lines of C code. Recently, application developers have made a transition. Where we might have had 100,000 lines of C code before, we might have 5,000 lines of C++ code now that make calls either into custom controls, VBX controls, or dynamic link libraries. What's just beginning to emerge now are applications like that, but additionally they can expose some of their data and their functions to other applications. Those applications can either be written in C or C++, but most commonly right now, this is done with Visual Basic. I know Basic's probably a dirty word to you, but that's what Bill Gates wants and I believe that's what he will get. Visual Basic for Applications (VBA), in one form or another, will be in all the Microsoft Office products. It's in Visual Basic itself now, it's in MS Word now, it's in MS Project, and it's in Microsoft Excel. There are some minor differences, but virtually all of them are capable of driving other applications called OLE servers.

OLE is built on top of COM. COM solves four critical design problems in order for the industry to have reusable software. These four problems are interoperability, versioning, language independence, and something called transparent remoting—which gets me really excited about the future of actuarial models. Interoperability means components supplied by different vendors can interoperate safely now. COM defines a binary, not a software coding standard for this. The reason why binary standards are important is that software standards for code reuse just haven't worked out. C++ was supposed to trigger this huge wave of production by third parties of class libraries. We were all going to buy these class libraries and plug them into our code and everything was going to be great. The problem is that in this particular case, different C++ compilers, when you compile up the program, place different names in your object file. So, if you get a class library that was compiled with Borland, and you try and use it with your Microsoft compiled application, they can't find each other's functions; it just doesn't work. With COM, you can mix and match components basically at will, as long as they support the interface you're interested in. Versioning is essential.

For the first time, COM provides a standard that dictates how functionality can be added to existing components without breaking existing code. In fact, COM mandates this kind of backwards compatibility. Perhaps some of you had the experience in your actuarial models of writing a modification and then, when you get an update, all of a sudden your modification doesn't work any more. This is the type of thing that COM versioning is supposed to solve. With COM, you define a programming interface, which is a collection of functions and some data you can access through functions. If you want to modify that interface, you have to create a new interface and maintain the old interface as well. So new code, more recent application code, can find and call old code that uses the old interface into the new interface so it will work. Additionally, there's a type of library built into Windows where you will be able to query and find out exactly which interfaces are available in which objects. A COM component can be tied into the Windows help system, so you will get verbal descriptions of what these objects do, what the calling conventions are for the functions, what the data attributes are, and what they mean. It's really hot technology.

Language independence is required to create the market breadth necessary to support specialized component vendors. You will be able to write and employ reusable components in the language of your choice. I do have one caveat here: I don't know if or how they will implement this with APL interpreters yet. One language can call a function in a component that was written in another language because it's a binary standard, not a source code standard. So you can buy one reusable component which can serve virtually all your internal programming needs whether you're using Visual Basic, C++, COBOL, Access, Excel, Delphi, or MS Word: all of these will be able to use these reusable components. The components that you can get are really amazing. I'll mention just one. You can buy an Excel clone for about $250. It's a perfectly good, working copy of Excel 4. It was developed by the same people who developed the spreadsheet called Wings, which was a leading spreadsheet on the Apple Macintosh. You can embed that in your application and redistribute it royalty free! Unlike the C++ reusable class library market, which largely fizzled, the component market is just burgeoning.

Transparent remoting is a technology that I think is really slick. There are three types of COM servers; and again, a server is a program or component that offers up functions and data for use by other programs or components. The first is the in-process server, which is a component server that exists in the same address space as the client application that's trying to use its services. Typically, these take the form of a DLL. That's the kind of server with the fastest response time.

The second type is called a local server. A local server is a component or application that exists as another process on the same machine as the client application or component. A local server typically takes on an EXE format; this is the type of server you are using when you control Excel or MS Graph from MS Word, for example.

The third type of server, the remote server, is the one that's really amazing. A remote server is a component that executes on another machine on the network. Your client application code for all three cases is identical. Your client does not know if the COM server is in process, local, or remote. Other than for reasons of performance, you really don't have to care. The server could be in your address space, it could be in another EXE, in another Windows in a separate address space, or it could be across the network. The server could even be running on a Unix machine, because remote COM servers are built upon something called the Open Software Foundation's (OSF) distributed computing environment's (DCE) remote procedure call (RPC) specification. What this means is that when you want to pass values over to one of these functions, there's some code underneath the surface, provided by the operating system, that performs what's called marshalling. It takes all of your data, bundles it up, ships it over the wire to its counterpart, puts it in the right address space, sets it up, and makes the function call. The server might be on a Unix box, it might be on an Intel NT box, or a DEC Alpha NT box, or in a separate process on your local machine. In each case, the return results come back, get marshalled, and are sent back across the wire or across local address spaces to your client process. It's much slower than an in-process server, but it works.

When the number and size of the parameters and return results are small, when the service is called infrequently but is very time intensive, or when the server is a faster platform, your performance can actually improve. If the same service is provided by multiple servers on the network, then you really can get dramatic results via remote parallel processing. I believe COM and OLE will open up a whole new generation of distributed processing applications for actuarial software, that will be remarkable. Of course, existing vendors would have to make a few changes to their architectures, which reminds me of my one joke for the day. Does anybody know why God could create the Heavens and the Earth in just seven days, while it takes your software vendor so long to simply get out his next release? Well, God didn't have an installed base! If you have ever developed commercial software, I am sure you can appreciate both the humor and the kernel of truth in this. OLE automation is built on top of COM. It's a little higher interface to do the types of things I have been talking about. It allows a macro programmer, whether in Excel or Visual Basic, to use and control another application.

As I mentioned, OLE automation controllers are applications that use the services of OLE automation servers. The reason to use OLE automation is you don't have to reinvent the wheel. You can use the best of breed component in your application, which keeps you free to focus on what you do best. If you market a commercial application, and if you expose your internal functionality and data through COM and OLE automation, you're opening up your product so third parties can come in and add value to your application; every line of code that they write adds value to your product. And lastly, your product will be much more standardized from the perspective of your end users. Did any of you follow Steve Strommen's work on defining a standard mortality database for the Computer Science Section? What Steve did was to define an ASCII files format for a mortality table database and a binary file format. He posted a note saying it would be nice if somebody could find a way to get these rates in and out of spreadsheets. So Peter Gerritson and I took what Steve did and created an OLE automation server application. It's a Windows EXE. We will post it on the Society's bulletin board for people to play with as soon as Steve finalizes his file formats.

I'd like our Computer Science Section to seriously consider the following proposal. I think the actuarial modeling domain is generally well-defined at this point. It's not like it was five, six, or seven years ago when we were all trying to learn how to deal with interest-sensitive products, and we didn't know what kinds of studies we wanted to perform, or what types of instruments we were going to run into, or how to get the assets and liabilities linked together. I believe that it's now possible to publish a purely virtual interface for actuarial model servers. By a virtual interface, I mean a defined collection of data variables or properties and functions or methods that any actuarial modeling system should have, piece by piece, component by component, without stipulating at all how any of these things should be implemented. This specification would dictate that certain functions and variables would have to exist, but would be silent on how they would be implemented. If we do this, we could finally get to the point where we could intermix pieces of software from our different actuarial vendors. If you like the Tillinghast Actuarial Software (TAS) CMO model better than you like the Profit Test System (PTS) CMO model, you ought to be able to just plug in and replace that with PTS. If you want to plug in your own, you ought to be able to do that too. If you want to take an entire asset model at a high level of its interface, and replace it with something that reads precalculated values from files as opposed to calculating them on the fly, you ought to be able to do that. I think it's something that's a very doable project. It would take a while to do it, but I think the benefits would be well worth the effort.

I think we would all benefit from increased competition and specialization amongst the providers of actuarial software. We'd get some enhanced consistency and reliability between our models, users could selectively use and more easily modify components, and users could more easily create one off specialty applications, like illustration prototypes on new products. In other words, if you've got some special project that doesn't quite fit into the framework of your large modeling system or your valuation system, and if these systems are written in a way that they expose their objects and methods through OLE automation, you ought to be able to pick and choose the components you need from your existing software, and then get into something like Excel or even Microsoft Word, believe it or not, and write a program to do what it is you need to do. It would greatly reduce actuarial training costs. Actuaries could learn one abstract modeling interface, and carry that knowledge with them from system to system and from company to company.

My final thought for the day is a piece of free advice, especially for anybody that's interested in software development and is younger than me. I recommend you forget about APL and C++. The way Windows is evolving, meaning the Microsoft Office Suite of products, it appears that Windows 95 or Windows NT or Cairo will not be the operating system of the future, and Visual Basic for Applications (VBA) is its Application Programmer's Interface (API). I should repeat that: Microsoft Office is the operating system and VBA is its API. This idea isn't original to me, but it's true, and many software developers share its sentiment. There is so much functionality being built into the Microsoft Office Suite that there's really not much room for anyone else to come out and develop word processors or spreadsheets or presentation packages. Microsoft has positioned these products as building blocks for vertical, commercial and in-house software development. Beginning with Windows 95, all of these products will be capable of acting as both OLE automation servers and as OLE automation controllers through a common programming language: VBA. That's the way you ought to view the future of corporate computing. The premiere way, the absolutely best way to exploit this development, is with Visual Basic for Applications. This is a conclusion I come to reluctantly, but I believe it's a valid one.

FROM THE FLOOR: What does Lotus say about that; will it have the VB interface?

MR. DAVLIN: Not at the present time, but it may be forced to come along. Microsoft does not license an embeddable VBA interpreter, but there are already three other vendors who have created VBA clone interpreters which support OLE automation, and they come complete with VBA debuggers that you can license and redistribute.

MR. SMITH: I have a couple comments dealing with large quantities of data, issues on backups and different things that I have seen recently after developing it or after researching it. I want to talk about some things we are seeing and doing in terms of the data structures or systems that are out there.

Central file servers can have extremely large amounts of data, something I'll call an actuarial server. This is something we're getting into. It used to be high-performance PCs intended to work on stand-alone operations, large disk capacities, for example. We have a number of them in our offices, 10–20 gigabytes a piece on a local PC where we do special jobs, special analyses. The other type of situation we see out there is individual PCs on your desk in a personal product, that's where you do your work up to one gigabyte. Something happened to me recently in what I'm calling actuarial servers. The disk just died halfway through a project. It made a couple of strange noises and then stopped making noises altogether. Well, wouldn't it have been pleasant to have had some backups. Now when I got into some backup issues, there's something called a complete image copy. Our server had one of those, and this was a central server, we could make complete image copies. Everything was dumped back out there, but with very little control on what files you might want to restore or move up. Some things, in terms of the ability to complete a project and save that data, were difficult to do. I investigated what was possible and found out we could have a disk system and back up on disk. As long as we have two copies on disk, if one of them dies, you have a spare. That didn't seem to be feasible.

I wanted to find out more about this—for example, the various costs, the functionalities and how long these things take. There's a certain class of units out there. Eight millimeter tapes tend to be bigger units. The cost of these is somewhere between $2,000 and $5,000. The capacity in terms of the tapes with some compression is 3.5–35 gigabytes. I looked at the media cost, and it might take you anywhere from one-and-a-half to three hours. There are different vendors out there, but that's roughly the time it would take. So if you started it after lunch, it might be done by the time you go home. You wouldn't want to be holding the elevator while this thing finished. There's something out there called four millimeter tapes. The cost of these is a little bit less. The media capacity was comparable, although it didn't really get into the high end. Some of these units you can stack, some eight millimeters seem more geared for the high-end server or central server things. The thought of spending $1,500 or $2,000 to add a tape unit onto my PC that didn't cost a whole lot more than that was less than fulfilling. There are mainframe style tapes, the square cartridge tapes, and they are handy for transferring data. Especially if you get something off your mainframe, you can just carry the tapes around or send them around. The cost of the units tend to be quite a bit more. They are also quite large. The performance is OK, but this really wasn't what I was looking at here.

Something just came across my desk called quick or QIC tapes. I'm not sure exactly what that means or if it's a class of tapes. I found one here that looks like the kind of thing you need for that one big machine. It has a lot of disks, a lot of capacity, and costs about $500 for the unit itself.

I also found a DLT-type of tape unit, Digital Linear Tape. The units have media capacity, twin gigabytes, and there are several models up to 140 gigabytes. A couple of years ago we bought the small portable units you plug into the parallel port. You get a couple hundred megabytes on a tape, and you think about trying to backup 20 gigabytes. By putting 200 megabytes on a cartridge, you would not be able to fit all the tapes in your pockets. So the technology is definitely improving and the media costs are really not that bad.