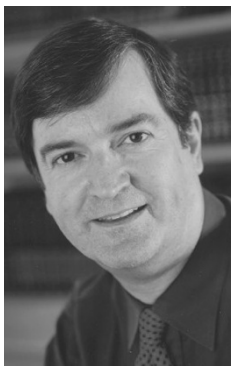
The associative model of data

Received: 24th May, 2001



Simon Williams

was founder, Chairman and Chief Executive of Synon Corporation, a worldwide leader in application development technology, from its foundation in 1984 until 1991. Simon conceived and developed Synon/2, the dominant development environment for IBM's AS/400 platform, Obsydian, an award-winning application development tool and Synon/Financials, a successful financial application package.

Following Synon's move to the USA in 1990, Simon remained in the UK as Group Chairman and in 1991 left to form Dysys, a software start-up that built the Obsydian product. In 1992 Synon acquired Dysys and Simon rejoined Synon as Chief Technology Officer until the end of 1996. In 1998 Synon was acquired by Sterling Software (itself recently acquired by Computer Associates) in a share swap valuing Synon at $80 million.

Abstract Lazy Software has created the Associative Model of Data,™ the first entirely new database architecture since the advent of the Internet. Its product Sentences™ is a multi-user, web-enabled database management system written in Java, and is the first commercial implementation of the Associative Model.

Using Sentences, customers can design and develop sophisticated database applications more quickly and with less technical know-how than has previously been possible. The high cost of application development has forced many companies to adopt packaged solutions that are not well tailored to their needs. Sentences brings custom applications back within their economic reach.

INTRODUCTION

A database management system is a vital component of every computer. At the heart of every computer system lies a vital software component called a database management system that stores and manages data. Database management systems range in scale from the simple file manager found on every PC to highly complex systems capable of storing huge volumes of data and affording simultaneous access to thousands of users. According to Dataquest, $8bn-worth of database software was sold in 1999, representing 18 per cent growth over the preceding year. The market leaders are IBM and Oracle with about 30 per cent of the market each, while Microsoft has about 10 per cent, and Sybase, Informix and CA each have 5 per cent or less.

Relational database technology is dominant today. The market comprises three sectors: relational databases, which account for the lion's share of size and growth; older pre-relational databases, which is a declining maintenance market focused on IBM mainframes; and object databases, the newest sector, which has been slow to develop and was worth less than $200m in 1998. The relational model of data was first proposed in 1970. Its commercial potential was recognised during the 1980s, and today it is dominant and ubiquitous to the point where virtually every modern data-processing application relies on a relational data base.

Multimedia first exposed the relational model's limitations. Networked PCs and the Internet have replaced mainframe systems as the focus of the industry's attention, and the role of computers has expanded from its traditional base of transaction processing into new areas, many of which depend on the PC's

Simon Williams
Lazy Software Ltd, Mercury Park, Wycombe Lane, Wooburn Green, Bucks HP10 0HH, UK.

Tel: 01628 642300; e-mail: info@lazysoft.com

multimedia capability. While the relational model is well suited to transaction processing, it cannot manage the complex data structures such as image and sound which are typical of multimedia applications. Also, the relational model does not easily support the distribution of one database across a number of servers, which is the natural model for the Internet. Furthermore, new applications are emerging that are beyond the relational model's capability, such as those dealing with spatial and temporal data, and with uncertainty and imprecision.

The object model failed to challenge the relational model's dominance. The object model of data evolved during the 1980s as an adjunct to the development of object-oriented programming languages, and a new generation of database products based on the object model started to reach the market in the early 1990s. Its proponents argue that the object model is the natural successor to the relational model, but sales of object databases have fallen far short of analysts' predictions and the market is not growing. It can be argued that in time the object model will overcome its difficulties and supersede the relational model: after all, the relational model itself took over ten years to succeed. There are good reasons why this will not happen. The object model was not originally conceived to improve on or even compete with the relational model, but was intended simply to provide persistent storage for object-oriented programming languages. As a result, in many respects it is inferior to the relational model for transaction processing. Also, while many new applications demand more granularity (the ability to deal individually with very small pieces of data) than the relational model can provide, the object model actually supports less granularity than the relational model, not more. Moreover, the object model predates the explosive growth of the Internet, and, in common with the relational model, is not well suited to meet its unique demands.

The market has not embraced object/relational products. In response to the perceived threat of object databases, the established vendors of relational databases began to incorporate some features of the object model into their relational products, creating hybrid 'object/relational' databases. Informix was first with the high-profile launch of its Universal Server in 1997, but it had misjudged the market's readiness to adopt its new technology, and the introduction was commercially disastrous — in the first quarter of 1997 its licence revenue fell by 50 per cent. Some commentators argue that object/relational represents the next major data model. But inspection of these products reveals that they are simply relational databases with a few of the features of the object model grafted on, and it is difficult to construct an effective economic case for the adoption of this technology. The addition of object features to relational products is best viewed as a mid-life kicker for the relational model.

Relational databases have not adapted to the internet. The Internet places a new set of demands on database technology to which relational vendors have yet to respond. Their flagship products were developed before the advent of client/server and the Internet, and have become feature-rich to the point of redundancy. By contrast, the Internet demands compact, lightweight technologies developed using Java that can function equally well on PCs, servers, network computers and the new generation of pervasive computing devices that is now emerging. One major vendor's Internet strategy requires companies to store their databases on large, centralised data servers, an

approach which is at odds with the distributed nature of the Internet. This defensive strategy is shaped more by the limitations of technology that the needs of customers, and highlights the relational model's inability to respond to the demands of the Internet.

The associative model overcomes key limitations of the relational model. The most visible limitation of the relational model has been its inability to handle complex data, but the market's verdict is that the importance of this has been exaggerated. Looking beyond complex data, the relational model has some far more significant limitations that the market has not yet challenged, and that are overcome by the associative model.

Using the relational model

Brains do not need new thought processes to think about new things — why do computers? Every new relational application needs a new set of programs developed from scratch, because a program written to use one set of tables cannot be reused with a different set. This creates a need for a never-ending supply of new programs, the development and maintenance of which is labour-intensive, expensive and wasteful. As long as marketers continue to rely on the relational model, application software will be far more costly than it needs to be.

Users do not all need the same functions — why is customisation so difficult? Relational applications offered by ASPs (Application Service Providers) and package vendors can only be tailored to the needs of large numbers of individual users through complex parameterisation or through customisation which renders subsequent upgrades more difficult. Finding a way to support the customisation of applications for individual users is one of the main challenges faced by early players in the burgeoning ASP marketplace.

All customers are not the same — why store the same information about each one? Relational applications cannot record a piece of information about an individual thing that is not relevant to every other thing of the same type. Consequently, applications have to store the same information about every customer, order, product and so on. This limits marketers' ability to continually improve the quality of customer service, because applications cannot record and take account of the needs of individual customers. The relational model is a limiting factor in the quest to improve customer service and competitive edge.

All databases store data — why can they not work together more easily? Information about identical things in the real world is structured differently in every relational database, so it is difficult and expensive to amalgamate two databases. The cost of integrating systems is now a major impediment to mergers and acquisitions. Extracting useful information from across several databases demands expensive data warehousing and mining projects.

Using the associative model

It is no longer necessary to write every new application from scratch. The same set of programs can be used to implement many different associative applications without being altered or rewritten in any way, allowing users to create new applications from existing ones. The saving in software development costs afforded by this capability will be substantial.

Applications can be tailored for individual users. Associative applications can permit features to be used or ignored selectively by individual users without the need for parameterisation or

customisation. Data sets can be similarly partitioned with precise granularity, to be visible or invisible to individual users. This approach is ideally suited to the needs of ASPs and application package vendors alike.

The information needed about each customer can be stored precisely. An associative database can record information that is relevant only to one thing of a particular type, without demanding that it be relevant to all other things of the same type. With this capability, we can continue to enhance the quality of customer service and hone competitive edge.

Databases can be integrated without extra programming or data warehousing tools. Separate associative databases can be readily correlated or merged without extra programming, and multiple databases distributed across many servers can be accessed by applications as though they were a single database. These capabilities significantly reduce the cost of amalgamating databases, and allow information to be readily extracted from across multiple databases without the need for data warehousing.

DATABASES SO FAR

Databases were made possible by magnetic disk storage. Computers did not always have database management systems: early computers stored their data on punched cards, paper tape or magnetic tape. In order to retrieve some data from part of the way through a deck of cards or a reel of tape, the computer had to read past the intervening data first, rather like fast-forwarding a videotape to find a particular scene. By contrast, magnetic disc storage enabled the computer to retrieve any piece of stored data almost instantaneously, rather like selecting a particular track from an audio CD. Its arrival in the early 1960s prompted a flurry of research into how information could best be represented and stored in a computer. The various solutions that were proposed were called 'models'.

To date, five models of data have been developed and exploited commercially. The network, hierarchical and relational models emerged between 1965 and 1970. The relational model won the largest market share and became the standard, at the expense of the other two. The object model emerged in 1990, and the hybrid object/relational model followed in 1996 as an attempt to unify the object and relational models, but neither have seriously challenged the dominance of the relational model, which represents over 95 per cent of the market. A brief summary of the relational, object and object/relational models follows.

THE RELATIONAL MODEL

Tables, rows and columns

The relational model was described by Dr Edgar Codd of IBM's San Jose Research Laboratory in his 1970 paper 'A relational model of data for large shared data banks'. The relational model stores data in tables. Each table holds data about one particular type of thing: customers, products, orders, employees and so on. Within a table, each row represents one instance of the type of things that the tables stores — one customer, one product and so on — and each column represents a piece of information that is stored about every customer, product and so on. Thus a customer table might have columns for customer number, name, telephone number, credit limit, outstanding balance and so on. Simple examples of customers and orders are shown in Tables 1 and 2.

Within each table, rows are uniquely identified by one or more special

Table 1: Customers

Customer number	Name	Telephone number	Credit limit	Outstanding balance
456	Avis	020 7123 4567	£10,000	£4,567
567	Boeing	020 8345 6789	£2,500	£1,098
678	CA	0123 45678	£50,000	£14,567
789	Dell	0134 56789	£21,000	£6,789

Table 2: Orders

Order number	Date	Customer number	Item	Quantity
11234	2-Mar-99	567	ABC345	150
11235	15-Mar-99	789	GGI765	25
11236	21-Apr-99	789	KLM012	1,000
11237	7-May-99	456	GHJ999	50

columns called primary keys, shown in bold. The relationship between an order and the customer who placed it is recorded by putting the customer's number into the 'customer number' column of the order. This is an example of a foreign key. Foreign keys are generally shaded but are shown unshaded in these tables.

The meaning of 'relational'

Many people who hear the term 'relational' for the first time assume that it derives from relationships between the things stored in the database. In fact, it comes from the mathematical concept of a 'relation': 'Given sets S1, S2, . . ., Sn, R is a relation on these n sets if it is a set of n-tuples, the first component of which is drawn from S1, the second component from S2, and so on'. The proper term for the tables described here is 'relations'.

Why must new programs be written for every application?

Brains do not need new thought processes to think about new things — why do computers? Every new relational database application needs a new set of programs written from scratch, because a program written for one application cannot be reused for another. This creates a need for a never-ending supply of new programs, the development of which is labour-intensive, time-consuming and expensive. Why is this so?

Programs are designed around tables. Under the relational model, every table is structured differently — that is, it has different columns and column headings — and the programs are designed around the tables. It is impossible to write an efficient program that is capable of accessing a table whose structure is not known when the program is written, just as it is impossible to make a key that will open any lock. Every program has to be written by someone with precise knowledge of the tables that it will use, and a program that uses one set of tables cannot be used with a different set. In commercial applications, each entity type — customers, products, orders and so on — is represented by at least one table, and most applications involve between 50 and 500 entity types, so each new application needs somewhere between 500 and 5,000 new programs to be

written from scratch. Even using modern 4GLs (and development tools), a non-trivial program can still take days or weeks to develop.

Most programs are hand-coded from scratch. One of the stated goals of object-oriented programming was the re-use of program code. Some 20 years after the first object-oriented languages were developed, however, almost no true re-use has been achieved. Some development tools automate the process of writing programs by re-using program designs, but such tools demand higher levels of skill and training and thus greater up-front investment than traditional programming techniques, so despite their impressive productivity levels their use is not widespread. Most programs developed today are still hand-coded from scratch in a highly labour-intensive manner. Re-use has failed not because programming languages and tools are deficient or because programmers are not clever enough, but simply because data are not stored in a way that permits it.

Why is it expensive to customise applications for individual users?

Not all users need the same functionality. The cost of software development has led more and more companies to depend on packaged applications and, increasingly, on Application Service Providers (ASPs). ASPs host applications and data on their own servers, which are accessed by their customers via the Internet. Both package vendors and ASPs face the same challenge: how to provide the richness of functionality required by sophisticated users without overwhelming those with more prosaic needs.

Parameterisation fuels complexity. Historically, the behaviour of each installation of an application package has been determined by parameterisation: the application checks a set of parameter values as it executes to determine precisely how its code should behave. This approach has drawbacks arising from the exponential increase in complexity as new options are added over time. The code itself becomes extremely complex: different pieces of business logic need to check different parameters to determine whether they are invoked, and as the number of options increases, new functions are more difficult to add, and testing the full range of configurations created by different combinations of parameters becomes more difficult. Also, deploying the package becomes very costly for customers. The lion's share of the cost of installing a sophisticated package goes on the specialist assistance needed to implement it. A major component of this is the time and know-how involved in setting up the package to achieve the desired behaviour.

Modified packages are difficult to upgrade to new releases. Users who require functionality not provided by the core package must modify their copy to create the behaviour that they require. This greatly increases the difficulty of upgrading to new versions of the package provided by the vendor, which often contain important new functionality that the customer would wish to exploit. A small industry of source code comparison, configuration management and impact analysis skills and tools exists to cater for precisely this need, but even so, typically fewer than 50 per cent of major application package users implement new releases for this reason.

The ASP model adds a new dimension to the problem. The advent of the ASP model adds a new dimension to the equation. When each customer has their own version of the package installed on their own computer, the

problem is confined to making one isolated installation of a package behave in one particular way. But the ASP must host the package for every one of its customers. If an ASP has, say, 10,000 customers using a package, does this mean it may find itself hosting up to ten thousand copies of the package as each customer demands their own configuration? If this, or anything approaching it, turns out to be the case, then the set-up and management costs alone would rapidly overturn the pricing assumptions that make the ASP model viable.

Applications can't be tailored to suit individual users. Parameterisation and customisation is a viable route, albeit expensive and cumbersome, for package installations where all users are required to use the same functionality, but what if there is a business need to tailor the behaviour of the application for an individual user? For example, many salespeople have their own ways of getting closer to their key customers: some wine and dine, some remember family details, some play sports. Nevertheless, the cost of recording customers' favourite cuisines, children's birthdays or golf handicaps in a company's enterprise database is unlikely to be deemed acceptable.

Why must the same information about every customer be stored?

The relational model cannot record or process a piece of information about an individual customer which is not relevant to every other customer, so the same information is stored about every customer. This limits companies' ability to improve continually the quality of their customer service, because systems cannot record and take account of the needs of individual customers. Why does such a fundamental restriction exist?

Every column of every row must be occupied. The relational model dictates that every row must have a value in every column. If the value is unknown, the column must contain a special mark called a 'null'. For a relation to store a piece of information that relates to only one row, an entire column in every single row would be have to be set aside to cater for it, and the empty column in every other row would have to contain a null. In commercial applications, it is not unusual for a relation to contain many thousands or even millions of rows. If a customer table contained 10,000 customers, setting aside a column to record a piece of information unique to one customer would entail storing 9,999 nulls. To store information unique to just 10 per cent of customers would add 1,000 columns to the table, and entail storing almost 10m nulls. The overhead of storage space and processing time to do this renders it impractical.

Only a programmer can change what information is stored. Nor do the problems end there. Even if a marketer decided to accept this overhead in the interests of customer service, there is another issue. If a salesperson needs to record a new piece of information, they cannot simply add the new information to the customer's record on the spot. A programmer needs to amend the table's definition, and then change every program in the system that needs to use the new piece of information. Again, this is clearly impractical.

'Market of one' is tough to implement using the relational model. Customer relationship management, and the build-to-order approach known as 'mass customisation' or 'market of one', are hot issues. To implement these techniques properly, companies will need computer systems that can recognise customers' individual needs. At a time when investment is shifting from back

office to customer-facing systems, the relational model is limiting marketers' ability to improve customer service further by insisting that systems record the same type of information about every customer.

Why is it so difficult to combine and correlate databases?

Functionally similar relational databases are always different. Suppose two systems analysts separately designed two databases to solve the same problem. The two databases would differ in many ways. They would contain different tables with different names, and even tables that did the same job would have different columns with different names in a different order. Next, suppose that the two databases were allowed to operate independently for several years — perhaps in two subsidiaries of a multinational — and then it was decided to amalgamate the subsidiaries and combine the databases. What would be involved?

It is impractical to merge tables from two databases. The simplest case involves two tables that perform the same function: such as the two customer tables. The rows cannot simply be added from one customer table to the other, because every row in a relation must have the same columns, and inevitably there will be at least one pair of columns that do not match. So both tables have to be examined and the corresponding columns matched up. Even when columns whose functions match are found, often they will contain different types of data: one designer may have chosen to identify customers by a number, and the other by a code using both letters and numbers. One must be chosen, new values assigned to the other, and then trawled the entire database through replacing old values with new ones. All this work deals with just one column in one pair of matching tables, but the nature of the relational database design process means that many of the tables in one database will have no direct equivalent in the other, so the process just described will often be the tip of the iceberg. Even when marketers stop short of merging databases — perhaps only a simple question needs answering, such as how many customers the two subsidiaries share in common — marketers have to go through this cross-referencing exercise before they can begin to find the answer to the question.

Distributed database capabilities do not solve the problem. Most database management systems incorporate facilities to distribute databases across many computers. They can put some tables on one computer and others on another, or some rows of a table on one computer and others on another, or some columns of a table on one computer and others on another. The distributed database can be administered either as one database stored on many computers, or as separate, unconnected databases. But this does not help overcome the problems. If the network is administered as one database, no benefit is gained in return for the overhead other than some leeway in resource utilisation. If it is administered as separate databases, the marketer is right back where they started. Distributed database capabilities typically solve only tactical problems, such as allowing one database to access a table that already exists in another. These limitations have forced at least one major database vendor to base their Internet strategy on centralised data servers.

Data warehousing is expensive and information is not always current. It is issues like these that have spawned the data warehousing industry, which provides tools and techniques that extract information from many databases and

gather it together into a single, cross-referenced central database that can be used for query and analysis. But a data warehouse is costly to set up and maintain, in terms of specialised skills, software tools and extra hardware to duplicate the data. Also the process can be time consuming, sometimes resulting in information that does not reflect the most recent trends.

THE OBJECT MODEL

Object orientation is a better way to write programs. Object orientation evolved during the 1980s as a better way of programming, superseding the prevailing philosophy of structured programming. In object-oriented programming languages, each piece of data in a computer is kept in the custody of an object, and cannot be directly accessed at all. Any program that needs to read or alter a piece of data must do so by sending a message to the custodian object, asking it to effect the desired operation by invoking one of its methods. Each method represents a process that an object can perform on its data. The data items that an object has custody of, and the methods that it responds to, are determined by the object's class. Every object belongs to, or to use the proper language, 'is an instance of' a class, and each class may have many member objects.

Bank accounts are object-oriented. A bank account is a good analogy for an object. A bank customer cannot change anything about their bank account directly: instead they send it messages in the form of cheques, deposits, balance inquiries and so on. A bank account's methods are 'pay a cheque', 'receive a deposit', 'produce a statement' and so on. A particular bank account is an instance of the class 'bank account'.

Object databases provide persistent storage for program variables. The object model of data was originally developed to provide persistent storage for CAD programs written using object-oriented programming languages. While an object-oriented program is running, all of its variables (that is, the data items that it is using at a particular point in time) are stored in main memory in the custody of objects. When the program ends, main memory is cleared and the objects are lost. Persistence is the capability that allows the objects to be stored on disk before the program ends, so that when the program is re-started, it can re-load its objects from disk and carry on exactly where it had left off.

A whiteboard is not persistent storage. A conference room whiteboard provides a good analogy for persistence. The content of a whiteboard is not persistent, because the next group of people to use the conference room will probably clean the whiteboard and write on it themselves. So if someone wants to keep a permanent record of anything written on the board during a meeting, they must copy it onto paper before leaving the conference room at the end of the meeting. Object databases were developed to be like the paper onto which the whiteboard's contents are written.

Object orientation came to be seen as a silver bullet. Object orientation was advocated with almost religious fervour as the solution to a wide variety of problems, ranging from a better way to write programs to a better way to re-engineer business processes. (Is a model that requires individuals and departments to hide their own information from all-comers and respond only to a limited and precisely-defined set of inputs really a good blueprint for modern business?) By 1990, the object model of data had come to be seen as a serious challenger to the relational model.

Industry commentators were quick to endorse the new contender, and big things were forecast for it, at the expense of relational technology. Some commentators still believe that, in time, the object model of data will supersede the relational model. The author disagrees. Here are the reasons.

Object databases have fallen far short of commercial expectations. When the object database market first emerged in 1990, it was expected to grow rapidly. In 1991, Ovum wrote 'DBMS that support objects will account for revenues of $4.2 billion in 1995 ... (and) object-oriented DBMS will earn revenues of $560 million in 1995'. In fact, Informix, the relational database vendor who has been most aggressive in supporting objects, has been severely rebuffed by the market, and object-oriented databases in 1995 achieved revenues of only $115m, about one fifth of Ovum's forecast. Early predictions for object databases assumed that they would win market share on two fronts: from their relational counterparts as the technology for storing transactional data, and from file systems such as Netware and DOS/Windows as the technology for storing multimedia data. In practice, neither has happened. For transactional data, vendors have held users' attention with developments such as data warehousing and data mining, while object databases continue to suffer from the perception of poor performance. For multimedia, few users have yet outgrown the capabilities of existing file systems to the point where an alternative solution is essential.

The object model was not conceived as a replacement for the relational model. The object model of data was not originally conceived to improve on or even compete with the relational model on its home turf, but was intended simply to provide persistent storage for object-oriented programming languages. Consequently, in many respects the object model is inferior to the relational model for transaction processing, most notably in view mechanisms and query processing.

Many basic object orientation (OO) concepts are not relevant to transaction processing. The original proponents of object orientation came from scientific and engineering disciplines rather than commercial data processing, and had had little exposure to the needs of transaction processing systems where relational databases excel today. Object orientation is first and foremost a better way to write programs. In the context of transaction processing, object databases are burdened with some irrelevant concepts. One such is encapsulation, whereby each data item is in the protective custody of an object and may be changed only by submitting a procedural request to that object. This is a good way to ensure the integrity of variables in a multitasking environment, but it is irrelevant to a general ledger table containing several million rows of data, and it makes the process of querying data more cumbersome and inefficient than it need otherwise be.

More granularity, not less is needed. Features such as encapsulation mean that, in an object database, there is a processing overhead associated with storing every object, regardless of its size. So, in practice, object databases store small numbers of large objects more efficiently than large numbers of small ones. This means that they promote less granularity than relational databases, not more. To improve, however, on the relational model in areas such as schema flexibility, querying, meta-data management, versioning, temporal and spatial data management, and long-duration transactions, more granularity is needed, not less.

The object model is too heavy for the

Internet and too complex for developers. Like the relational model, the object model was conceived before the explosive growth of the Internet. The Internet favours lightweight technologies whose components can be distributed across the Internet itself and embedded in browsers and websites. In trying to adapt itself, and at the same time keep pace with Microsoft and compete with the relational model, object technology has become complex and heavy. As well as rendering it unsuitable for the Internet, this complexity has also hindered its acceptance. For many developers and technical decision-makers today, object technology is now too difficult to use.

Many object databases lack essential database capabilities. Many object databases are not yet fully mature as database management systems, lacking essential features such as SQL support, authorisation and access control, performance tuning, and interfaces to transaction monitors and other databases. Over time, the technology will mature, but the market may not be patient enough to wait.

THE OBJECT/RELATIONAL MODEL

'Object/relational' means many things to many people. The object/relational model is the most recent development in database thinking. In such a large market with such high stakes to play for, it was perhaps inevitable that the object/relational model would become many things to many people.

Object/relational technology is the relational vendors' response to object databases. First and foremost, the object/relational model is the relational vendors' response to the challenge of object databases. Most relational vendors have added to their products such features of the object model as they think will satisfy their customers' desire for things to be object-oriented, and repackaged the products as so-called Universal Servers. These products are described as object/relational, and some commentators argue that this represents the next major data model. But inspection of these products reveals that they are still relational databases at heart, with a few of the features of the object model grafted on.

Some authors have addressed the concept more seriously. Several authorities have attempted to be more rigorous in their definitions. In 'Object-relational databases: The next great wave', Michael Stonebraker, founder of database vendors Ingres (now part of Computer Associates) and Illustra, defines object/relational databases as those which support a dialect of SQL/3, and adds 'They are relational in nature because they support SQL; they are object-oriented in nature because they support complex data'. This definition has the virtue of being succinct: however, one suspects it would not meet with the approval of veteran relational advocates Chris Date and Hugh Darwen, who in 1998 presented the first formal specification of the object/relational model in their book 'Foundation for object/relational databases: The third manifesto'.

Date and Darwen are not convincing about the object side of the alliance. Date and Darwen reassert the continued relevance of the relational model, which they restate and, within limits, redefine. They propose to replace SQL by a more truly relational language, to replace the ill-defined concept of a domain with their well-defined concept of a scalar type, to rewrite the relational algebra, and to introduce type inheritance. Despite their assurance that they are interested in the applicability of object concepts to database management, one

has to search hard to find them in their book. They even pull off a nice piece of semantic misdirection by using the abbreviation 'OO', firmly ingrained into marketers' psyches as 'object-oriented', to mean 'other orthogonal', which is something else entirely.

Object/relational lacks a conceptual model. Various authors have rightly stressed the importance of a clean, sturdy and consistent conceptual model as the foundation for any application, and, because database management systems form the basis for many other applications, the need for a sound conceptual model in their case is more fundamental than ever. But, with the exception of Date and Darwen, the conceptual models underlying other interpretations of object/relational technology have strayed so far from their roots in both camps that they demonstrably fail to meet this criterion. But no-one is likely to implement Date and Darwen's proposals because one of its pillars is the abandonment of SQL, and in the minds of a large section of the marketplace, SQL *is* the relational model. It would be impossible to market the benefits of the relational model without SQL, and it would be impossible to implement Date and Darwen's proposals without abandoning SQL.

Object/relational is essentially a mid-life kicker for the relational model. Date and Darwen aside, the selective addition of object-oriented features to products whose hearts and souls are relational owes more to marketing than to sound conceptualising, and cannot legitimately be viewed as the birth of a new data model. Time will tell whether the market will embrace or reject object/relational technology. To win broad acceptance, its proponents must demonstrate that the marginal value that it adds to the relational model justifies the cost of its adoption. Given that the market has already held aloof from the object model, it is difficult to see exactly how this might be demonstrated. Until the market decides otherwise, the addition of object features to relational products is best viewed as a mid-life kicker for the relational model.

THE ASSOCIATIVE MODEL

The associative model of data is the name given by Simon Williams to the set of concepts, structures and techniques underlying the Sentences database management system. The associative model builds on a body of academic research that includes triple stores, semantic networks, binary-relational techniques and the entity relationship model. Williams has added several important and unique concepts. A provisional US software patent application for the associative model has been lodged.

The associative model sees information in the same way as the brain does: as things and associations between them. These associations are expressed through the simple subject-verb-object syntax of an English sentence — hence the name of the product. Here are a few sentences that fit the syntax of the associative model. The verbs are written in italics to make the composition of the sentences clear. (Some 'verbs' are actually prepositions but this distinction is ignored for the purposes of the model.)

The sky *is coloured* blue
Mary Murphy *is sister* to William Peters
Cows *eat* grass
Grass *is a* plant
Avis *has a credit limit* of 10,000
London *is located in* the UK

A sentence may itself be the subject or

Table 3: Customers

Customer number	Name	Telephone number	Credit limit	Outstanding balance
456	Avis	020 7123 4567	£10,000	£4,567
567	Boeing	020 8345 6789	£2,500	£1,098
678	CA	0123 45678	£50,000	£14,567
789	Dell	0134 56789	£21,000	£6,789

object of another sentence, so the associative model can express quite complex concepts:

(Flight BA123 *arrives* at 16:15) *on* Saturdays
The Bible *says* (God *created* the World)

The relational and the associative models are next shown side by side. One of the tables that was considered earlier is first shown again in Table 3.

Then, the sentences:

Avis is a Customer
Avis has telephone number 020 7123 4567
Avis has credit limit £10,000
Avis has outstanding balance of £4,567
Boeing is a Customer

Boeing has telephone number 020 8345 6789
Boeing has credit limit £2,500
Boeing has outstanding balance £1,098

. . . and so on.

Entities and associations

Real-world things are entities or associations. The associative model divides the real-world things about which data is to be recorded into two sorts:

— entities are things that have discrete, independent existence. An entity's existence does not depend on any other thing
— associations are things whose existence depends on one or more other things, such that if any of those things ceases to exist, then the thing itself ceases to exist or becomes meaningless.

For example:

— a person is an entity, while a person's roles as a customer, an employee, a spouse, a salesperson, a shareholder, a team member and so on are associations
— an enterprise is an entity, while an enterprise's roles as a customer, a supplier, a contractual party, a tenant, and so on are associations
— a consumer good, such as a car or a television, is an entity, while its various roles as the end product of a manufacturing process, a production schedule line item, the subject of a warranty agreement, and so on are associations
— a building is an entity, while its various roles as a corporate headquarters, a workplace, the location of assets are associations.

An association may depend upon another association: for example, a sales order may depend on a customer, which is itself an association. Similarly each line of a sales order depends on the sales order itself (see Figure 1).

Things, and their interactions with

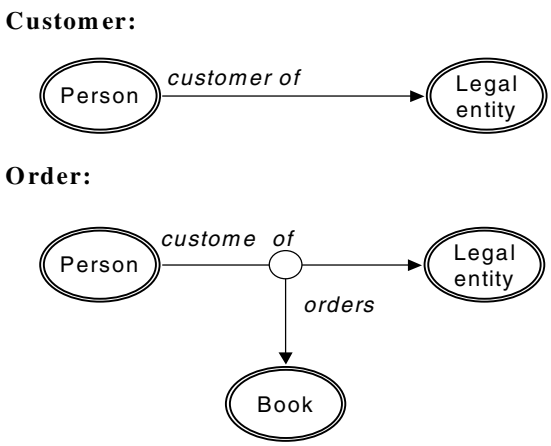


Figure 1

other things, are separate ideas. By classifying real-world things as entities and associations, the associative model separates two ideas: on one hand, the idea of something that has discrete, independent existence, and on the other hand the idea of the various ways in which such a thing interacts with other things. Each such interaction is a thing in its own right, about which marketers may want to record information. A real-world association is represented as an association between two other things, each of which might itself be an entity or an association.

The associative model distinguishes entities and associations for a simple and fundamentally important reason: data models constructed by following this principle are closer to reality, and thus easier to comprehend, better able to respond to change, and better able to integrate readily with other data models. Such data models will serve users better and prove more cost-effective, in both the short term and, more importantly, over the long term.

Most data models ignore this distinction. The distinction between entities and associations is one that other data modelling systems ignore or regard as peripheral. Most other systems would model a customer as an independent entity or object, while in the associative model it is an association. Specifically, the relational model does not distinguish entities and associations, on the grounds that both entities and associations have, in Codd's words, immediate properties. This is certainly a good reason to treat them similarly in many respects, but it is not a sufficient reason to ignore the distinction: to do so is rather like saying that because women and men are both human beings, therefore we can ignore any differences between them.

Associative building blocks

Items and links

An associative database comprises two data structures:

— set of items, each of which has a unique identifier, a name and a type.
— a set of links, each of which has a unique identifier, together with the unique identifiers of three other things, that represent the source, verb and target of a fact that is recorded about the source in the database. Each

Table 4: Items

Identifier	Name
77	Flight BA1234
08	London Heathrow
32	12-Aug-98
48	10:25am
12	arrived at
67	on
09	at

Table 5: Links

Identifier	Source	Verb	Target
74	77	12	08
03	74	67	32
64	03	09	48

of the three things identified by the source, verb and target may be either a link or an item.

How the associative model would use these two structures to store the piece of information 'Flight BA1234 arrived at London Heathrow on 12-Dec-98 at 10:25am' is now described. There are seven items: the four things 'Flight BA1234', 'London Heathrow', '12-Dec-98' and '10:24am', and the three verbs 'arrived at', 'on' and 'at'. Three links are needed to store the data. They are:

Flight BA1234 arrived at Heathrow Airport
... on 12-Aug-98
... at 10:25am

Each line is one link. The first link uses 'arrived at' to associate 'Flight BA1234' and 'Heathrow Airport'. The second link uses 'on' to associate the first link and '12-Aug-98'. (A link that begins with an ellipsis '...' has the previous link as its source.) The third link uses 'at' to associate the second link and '10:25am'.

When writing links, instead of using new lines to show each link, sometimes it is more convenient to keep going in a long string. When this is done, brackets are put around each link. Written this way, the example would look like this:

((Flight BA1234 arrived at Heathrow Airport) on 12-Aug-98) at 10:25am

This may look more like human language than the contents of a database, but if the marketer chooses for a moment to view the associative model in a relational sort of way, it can be seea that an associative database could be stored in just two tables: one for items and one for links. Each item and link has a meaningless number as its key (Tables 4 and 5).

THE BOOKSELLER PROBLEM

A more sophisticated problem that shows metadata as well as data is now considered. Here is the problem: An Internet retail bookseller operates through legal entities in various countries. Any legal entity may sell books to anyone. People are required to register with the legal entity before they can purchase. For copyright and legal reasons not all books are sold in all countries, so the books that each legal entity can offer a customer depend on the customer's country of residence. Each legal entity sets its own prices in local currency according to the customer's country of residence. Price increases may be recorded ahead of the date that they become effective. Customers are awarded points when they buy, which may be traded in against the price of a purchase. The number of points awarded for a given book by a legal entity does not vary with the currency in which it is priced.

Here are the metadata that describes the structure of orders. The items in bold are entity types.

Legal entity sells **Book**
... worth **Points**
... in **Country**
... from **Date**
... at **Price**
Person lives in **Country**
Person customer of **Legal entity**
... has earned **Points**
... orders **Book**
... on **Date**
... at **Price**

Now the data themselves. The items in italics are entities. First, the group of them being used are defined; two legal entities, two books, two customers and two countries:

Amazon is a **Legal entity**
Bookpages is a **Legal entity**
Dr No is a **Book**
Simon Williams is a **Person**
Simon Williams lives in *Britain*
Mary Davis is a **Person**
Mary Davis lives in *America*
Britain is a **Country**
America is a **Country**
Spycatcher is a **Book**

Next comes the price list:

Amazon sells *Dr No*
... worth *75 points*
... in *Britain*
... from *1-Jan-00*
... at *£10*
... in *America*
... from *1-Mar-00*
... at *$16*
Amazon sells *Spycatcher*
... worth *50 points*
... in *Britain*
... from *1-Jun-00*
... at *£7*
... in *America*
... from *1-Jun-00*
... at *$12*
Bookpages sells *Dr No*
... worth *35 points*
... in *Britain*
... from *1-Jan-00*
... at *£8*
... in *America*
... from *1-Jan-00*
... at *$14*
Bookpages sells *Spycatcher*
... worth *35 points*
... in *America*
... from *1-Jun-00*
... at *$13*

Now, for each of the two customers the number of points awarded to date are recorded, together with a single order:

Simon Williams customer of *Bookpages*
... has earned *1,200 points*
... orders *Dr No*
... on *10-Oct-00*
... at *£10*
Mary Davis customer of *Amazon*
... has earned *750 points*
... orders *Spycatcher*
... on *19-Oct-00*
... at *$12*

The metadata for the bookseller problem are shown in diagrammatic form in Figure 2. The ovals are items; the lines are links. The circles on the lines are the anchor points for links between items and other links.

The part of the data for the bookseller problem is shown in the same diagrammatic form in Figure 3.

Associative vs relational schemas

The metadata for the bookseller problem in associative and relational forms is now shown side by side. First, the associative metadata again:

Legal entity sells **Book**
... worth **Points**
... in **Country**

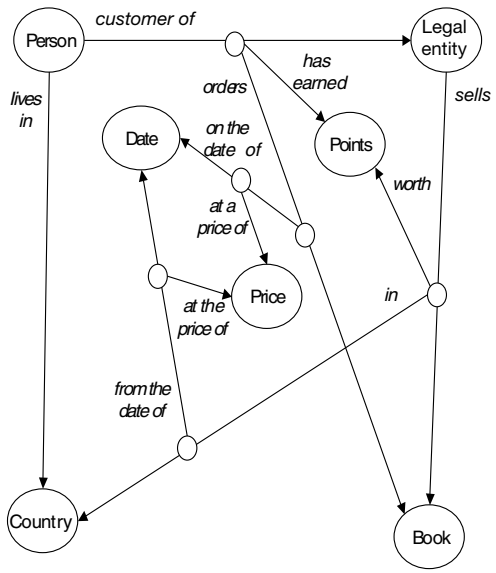


Figure 2

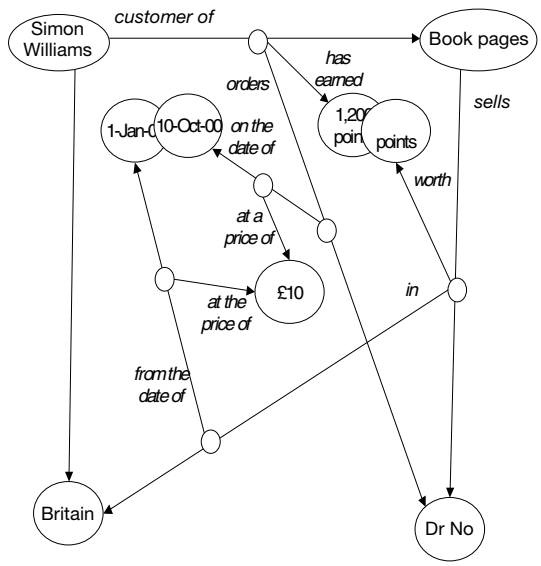


Figure 3

... from **Date**
... at **Price**
Person lives in **Country**
Person customer of **Legal entity**
... has earned **Points**
... orders **Book**
... on **Date**
... at **Price**

Here is the SQL solution to the same problem:

```
CREATE TABLE Person
 ( Person_id ,
   Person_name ,
   Country_id REFERENCES
     Country ,
   PRIMARY KEY ( Person_id ) )

CREATE TABLE Country
 ( Country_id ,
   Country_name ,
   PRIMARY KEY ( Country_id ) )
```

```
CREATE TABLE Book
 ( Book_id ,
   Book_name ,
   PRIMARY KEY ( Book_id ) )

CREATE TABLE Legal_entity
 ( Legal_entity_id ,
   Legal_entity_name ,
   PRIMARY KEY ( Legal_entity_id ) )

CREATE TABLE Books_sold
 ( Legal_entity_id REFERENCES
     Legal_entity ,
   Book_id REFERENCES Book ,
   Points ,
   PRIMARY KEY ( Legal_entity_id,
     Book_id ) )
CREATE TABLE
  Books_sold_by_country
 ( Legal_entity_id REFERENCES
     Legal_entity ,
   Book_id REFERENCES Book ,
   Country_id REFERENCES
     Country ,
   PRIMARY KEY ( Legal_entity_id,
     Book_id,
     Country_id ) ,
   FOREIGN KEY ( Legal_entity_id,
     Book_id )
    REFERENCES Books_sold )

CREATE TABLE Price_list
 ( Legal_entity_id REFERENCES
     Legal_entity ,
   Book_id REFERENCES Book ,
   Country_id REFERENCES
     Country ,
   Effective_date ,
   Price ,
   PRIMARY KEY ( Legal_entity_id,
     Book_id,
     Country_id, Effective_date ) ,
   FOREIGN KEY ( Legal_entity_id,
     Book_id )
    REFERENCES Books_sold,
   FOREIGN KEY ( Legal_entity_id,
     Book_id,
     Country_id ) REFERENCES
       Books_sold_by_country )

CREATE TABLE Customer
 ( Legal_entity_id REFERENCES
     Legal_entity ,
   Person_id REFERENCES Person ,
   Points_earned ,
   PRIMARY KEY ( Legal_entity_id ,
     Person_id ) )

CREATE TABLE Order
 ( Order_id ,
   Legal_entity_id REFERENCES
     Legal_entity ,
   Person_id REFERENCES Person ,
   Book_id REFERENCES Book ,
   Order_date ,
   Price ,
   PRIMARY KEY ( Order_id )
   FOREIGN KEY ( Legal_entity_id,
     Person_id )
    REFERENCES Customer )
```

So, what the associative model says in 11 lines of schema takes 51 lines of SQL. The relations that record the same data as the associative model example above are shown in Tables 6 to 14.

Table 6: Person

Person id	Person name	Country id
P123	Simon Williams	GB
P234	Mary David	USA

Table 7: Country

Country id	Country name
GB	Britain
USA	America

Table 8: Book

Book id	Book name
B345	Dr No
B456	Spycatcher

Table 9: Legal entity

Legal entity id	Legal entity name
L01	Amazon
L02	Bookpages

Table 10: Books sold

Legal entity id	Book id	Points
L01	B345	75
L01	B456	50
L02	B345	35
L02	B456	35

Table 11: Books sold by country

Legal entity id	Book id	Country id
L01	B345	GB
L01	B345	USA
L01	B456	GB
L01	B456	USA
L02	B345	GB
L02	B345	USA
L02	B456	USA

Table 12: Price list

Legal entity id	Book id	Country id	Effective date	Price
L01	B345	GB	1-Jan-00	£10
L01	B345	USA	1-Mar-00	$16
L01	B456	GB	1-Jun-00	£7
L01	B456	USA	1-Jun-00	$12
L02	B345	GB	1-Jan-00	£8
L02	B345	USA	1-Jan-00	$14
L02	B456	USA	1-Apr-00	$13

Table 13: Customer

Legal entity id	Person id	Points earned
L01	P234	750
L02	P123	1,200

THE ASSOCIATIVE EDGE

Four significant limitations of the relational model that the market has not yet challenged, and that the associative model overcomes were cited earlier on. Let us see how it does so.

One program, many applications

Every relational program must be designed around the tables that it uses. As seen earlier, each new relational application needs a new set of programs written from scratch, because a program written for one application cannot be reused for another. Every table is structured differently — that is, it has different columns and column headings — and the programs are designed around the tables. How does the associative model avoid this?

The information that describes how data are stored in a database is called 'metadata'. Metadata describe the structure and permitted state of data in a database. Structure is concerned with the different types of data that a database may contain, and how the different types of data interrelate. State is concerned with the rules which govern the values that data items may take, both individually and with respect to other data items. The metadata that describe a single database are called a schema.

In a relational database, a schema comprises the names of tables and columns and the domains on which the columns are based, information about which columns are keys, and 'referential integrity' rules that describe how some data items depend on others. The two

Table 14: Order

Order id	Legal entity id	Person id	Book id	Order date	Price
O2001	L01	P123	B345	10-Oct-00	£10
O2006	L02	P234	B456	19-Oct-00	$12

different parts of a relational schema are expressed in two different ways: everything except the referential integrity rules is expressed in SQL, and the referential integrity rules are expressed in a procedural language. Each vendor's system uses a different procedural language for this purpose.

Every program that uses a database needs a schema to tell it how the data in the database are structured. Programs obtain schemas in two ways: either the schema is known before the program is written and the program is designed to use the specific schema, or the program reads schemas as it goes and is able to interpret and act on anything that it finds. A program that is written to use one predetermined and unchanging schema is called 'unicompetent'. A program that is able to use any and every schema is called 'omnicompetent'. A good example of an omnicompetent program is a spreadsheet application such as Excel or Lotus 123.

It is difficult to write omnicompetent programs for relational databases. Relational databases comprise dissimilar tables each with dissimilar columns, and their schemas are stored separately, often using two different languages. This makes it very difficult to write omnicompetent programs, and there are no mainstream programming tools for relational databases that support the development of omnicompetent programs. With the exception of a few specialised tools such as report writers, the overwhelming majority of application programs that access relational databases are unicompetent.

By contrast, the associative model stores all data and schemas side-by-side in the same simple, consistent form. This means that it is easy to write omnicompetent programs using a form of abstract programming called 'metacode' that is part of the associative model. Metacode allows programs to be written that can operate on any and every business entity without modification. This substantially reduces the number of new programs needed for a new application. Also, as more applications are deployed, the proportion of new requirements that can be fulfilled by existing programs increases, so the number of new programs that have to be written decreases still further. Today, programmers continually have to re-invent the wheel by rewriting familiar programs to work with new tables. Breaking this cycle will significantly reduce the cost of computing.

The reusability of metacode means that many simple applications can be implemented using existing programs. This opens the door to a much greater involvement of end-users in the creation of applications. Once they become familiar with a core repertoire of Sentences programs, many end-users will be able to develop and deploy simple applications without any specialist help.

Different behaviour for different needs

Parameterisation and customisation are poor solutions for ASPs. Relational database applications can only be tailored to the needs of individual customers via parameterisation or customisation. The first route leads to increasing complexity, and the latter makes the upgrade path more difficult. Both options fail adequately to meet the needs of ASPs. The associative model provides a natural and unobtrusive way to solve these problems that does not compromise functionality or increase complexity.

Each user's view of an application can be tailored through a profile. An associative database comprises a number of chapters. A user's view of the database is determined by their profile, which

contains the list of chapters that they currently see. During the development of a schema, designers are free to place elements of the schema into any chapter that they choose, and that piece of the schema will be operative or inoperative depending on whether the chapter containing it is included or excluded from the user's profile. Similarly, during execution, changes that the user makes to the database may be directed into any of or several of the chapters in the user's profile.

Profiles are a powerful tool to manage functionality. Chapters may be added to and removed from a user's profile at any time, without deleterious effects on the integrity of the database. Of course, the user's view of the database may be radically altered by the addition or removal of chapters to or from their profile, and the ability to amend profiles is a privileged operation to which many users would not be authorised. For example, a human resources application to be used in both the UK and the USA might use three chapters for its schema:

Chapter A: **Person** first name **Name**
Seen by all users
Person family name **Name**
Seen by all users

Chapter B: **Person** postcode **Postcode**
Seen by UK users

Chapter C: **Person** zip code **Zip code**
Seen by US users

Profiles rely on granularity and change control. The profile mechanism works for two reasons:

— the associative model is highly granular. Individual items and links exist in peer networks in individual chapters. When chapters are collected together in a profile, the items and links in each chapter simply form a wider peer network, and chapters become transparent. When links are created between items in different chapters, if either the source or target of the link is not visible in the current profile, neither is the link itself
— changes and deletions are effected solely by additions to the database. A deleted association is not physically removed, but acquires an association (a 'stop link') which asserts that it is deleted. Thus the association may appear to be either deleted or not according to whether the chapter containing the stop link is part of the user's profile. Similarly, a renamed entity is not physically renamed, but acquires a link to its new name. Thus the new name or the old name is presented according to whether the chapter containing the new name is part of the user's profile.

Profiles have a variety of uses. Some scenarios that show how the profile mechanism can be used to advantage follow:

— an ASP wishes to provide unique functionality to each of its customers. As well as the chapters containing the application's core functionality, each customer's profile includes an extra chapter containing the schema changes unique to the customer. The core functionality can continue to evolve, and each customer's additions will continue to be superimposed. If customer A wishes to take advantage of functionality developed for customer B, customer B's chapter is simply included in customer A's profile

— an application is to be used by both English and French-speaking users. It is developed in English. A new chapter is added to the translator's profile as the recipient for all schema changes made by the translator. The translator then changes the name of every schema element to its French equivalent. Users who wish to see the application in French include the new chapter in their profiles: users who wish to see it in English omit it
— a salesperson using a CRM application wishes to record the birthdays of their key customers' children. A new chapter is created containing only the schema changes necessary to implement this functionality, and only the salesperson who requires the functionality will see it
— a package vendor is developing a new release of its package. The developers' profile includes the chapters containing the current release, plus a chapter for the new release into which schema changes are directed. To install the new release, customers simply add the new chapter to their existing profiles. To run the two releases in parallel for a test period, customers use a new chapter to capture data that instantiates types contained in the new release and work normally. They can flip-flop between the old and new releases by moving the two new chapters into and out of their profile to assess the impact of the new release.

Different information for different customers

All customers are not the same — why store the same information about each one? The relational model limits the marketer to storing the same data about each customer because it is uneconomic to set aside a column in every row to store a piece of information in just one row, and because a programmer needs to modify the programs every time a new column is added. How does the associative model overcome this?

Under the associative model, pieces of information about a customer are arranged vertically as a list of sentences instead of horizontally as the columns of a row in a table. So 15 sentences can be kept about one customer and ten about the next without implying that the second customer has five sentences missing. Unlike the relational model, there is no need to keep five null values for the five 'empty' columns: there are simply five fewer sentences. Addresses provide an example of how the associative model wins. Addresses may have one or two or three lines before city, state or county and zip or post code. In a relational database, the number of columns allowed for address lines is always a trade-off between clarity and waste. If three columns are allowed and a customer has only one address line, two nulls must be stored to say that the second and third lines are missing. If one column is allowed and the customer has three address lines, chances are their deliveries will go astray. In an associative database, each customer simply has the correct number of address lines: if a customer has two address lines, two are stored, and the third is not in any sense missing: it simply does not exist.

Users can capture new business rules without a programmer's help. The paper has already described how the associative model's metacode allows marketers to write omnicompetent programs that are able to read and use schemas as they go. When it is decided to store a new piece of information about a certain type of entity, the marketer simply adds to the schema. The need to modify programs each time a column is added then goes

away: the new information can be immediately understood and processed by the omnicompetent programs. For example, suppose a sales prospect, Avis, insists that, if to win its business, the system must guarantee that its account balance with the company will never exceed £10,000. No other customer has ever asked for such a guarantee, so it is not something that the system currently supports. The schema to define the new limit and its rules comprises just two new links which would have needed to be added:

Avis has balance limit Monetary value
... must not exceed Account balance

followed by one more sentence — this time real data, not metadata — to set the balance limit for Avis:

Avis has balance limit £10,000

The metacode in the programs already understands 'Monetary value' and 'must not exceed', so the task is done and Avis can be given the guarantee that it needs. In the relational world, an enhancement like this for a single customer would simply be uneconomic, so the Avis account would be lost. In the associative world, a moderately sophisticated user could readily use Sentences in this way to solve their problem and win a new customer without any need to involve a programmer.

Many databases, one data world

Associative databases may be readily combined. Combining two relational databases is like trying to combine two books written in different languages: before starting on the useful work one of them has to be translated into the language of the other. As seen above, before even beginning to combine two different relational databases matching tables and columns must be found and compared, differences resolved, and decidsions made about what to do about tables and columns that simply do not match. By contrast, combining two associative databases is like putting together two documents both written in English on the same word processor. One can immediately be added to the other with no preparation. The result will be perfectly comprehensible, and will answer more questions than did either text on its own. If the two databases are to remain as one, the user can then can edit and enhance the combined whole to establish common definitions and remove ambiguities.

Links can be added to resolve ambiguity. Associative databases can always be combined with no preparation because the associative model uses one consistent form — sentences — for all data and metadata. Every associative database has the capability of being self-defining: that is, of carrying its own definition within itself. Where two databases use dissimilar names — perhaps in different languages — for identical types of data or individual data items, users can associate equivalent types and data items to resolve ambiguity simply by adding extra links:

Customer *is equivalent to* Client
Delivery address *is equivalent to* Shipping address
BT *is equivalent to* British Telecom

Entity types that perform essentially the same function will usually have different sets of data items in each database, but, unlike the relational model, the associative model does not insist that all entities of one type have the same set of data items and does not allocate space for missing data items, so this is not an issue.

This capability of the associative model allows information in different databases to be correlated without the need for the additional costs of data warehousing, and permits separate databases to be readily combined. Individual databases and related networks of databases may also be distributed across networks and the Internet in any configuration without any administrative or programming overhead, allowing complete freedom of resource utilisation.